CPSC 330
Applied Machine Learning

Lecture 15: DBSCAN and Hierarchical Clustering

UBC 2022-23

Instructor: Mathias Lécuyer

Imports

In [1]:
```
import os
import random
import sys

import numpy as np
import pandas as pd

sys.path.append("../code/.")
from plotting_functions import *
from plotting_functions_unsup import *
import matplotlib.pyplot as plt
import mglearn
import seaborn as sns
from ipywidgets import interactive
from plotting_functions import *
from plotting_functions_unsup import *
from scipy.cluster.hierarchy import dendrogram, fcluster, linkage
from sklearn import cluster, datasets, metrics
from sklearn.cluster import DBSCAN, AgglomerativeClustering, KMeans
from sklearn.datasets import make_blobs, make_moons
from sklearn.decomposition import PCA
from sklearn.metrics.pairwise import euclidean_distances
from sklearn.preprocessing import StandardScaler
from yellowbrick.cluster import SilhouetteVisualizer

plt.rcParams["font.size"] = 16
%matplotlib inline
pd.set_option("display.max_colwidth", 0)
```

Announcements

- Homework 6 due tomorrow (Mar 15).
- Homework 7 will be released on Thursday (March 16).
- Final exam announced: April 20, at 19:00 PM (in CIRS 1250 or BIOL 1000, stay tuned for how we will use rooms).

Learning outcomes

From this lecture, students are expected to be able to:

- Identify limitations of K-Means.

- Broadly explain how DBSCAN works.
- Apply DBSCAN using `sklearn`.
- Explain the effect of epsilon and minimum samples hyperparameters in DBSCAN.
- Explain the difference between core points, border points, and noise points in the context of DBSCAN.
- Identify DBSCAN limitations.
- Explain the idea of hierarchical clustering.
- Visualize dendrograms using `scipy.cluster.hierarchy.dendrogram`.
- Explain the advantages and disadvantages of different clustering methods.
- Apply clustering algorithms on image datasets and interpret clusters.
- Recognize the impact of distance measure and representation in clustering methods.

Recap and motivation

K-Means recap

- We discussed K-Means clustering in the previous lecture.
- Each cluster is represented by a center.
- Given a new point, you can assign it to a cluster by computing the distances to all cluster centers and picking the cluster with the smallest distance.
- It's a popular algorithm because
  - It's easy to understand and implement.
  - Runs relatively quickly and scales well to large datasets.
  - `sklearn` has a more scalable variant called `MiniBatchKMeans` (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html) which can handle very large datasets.

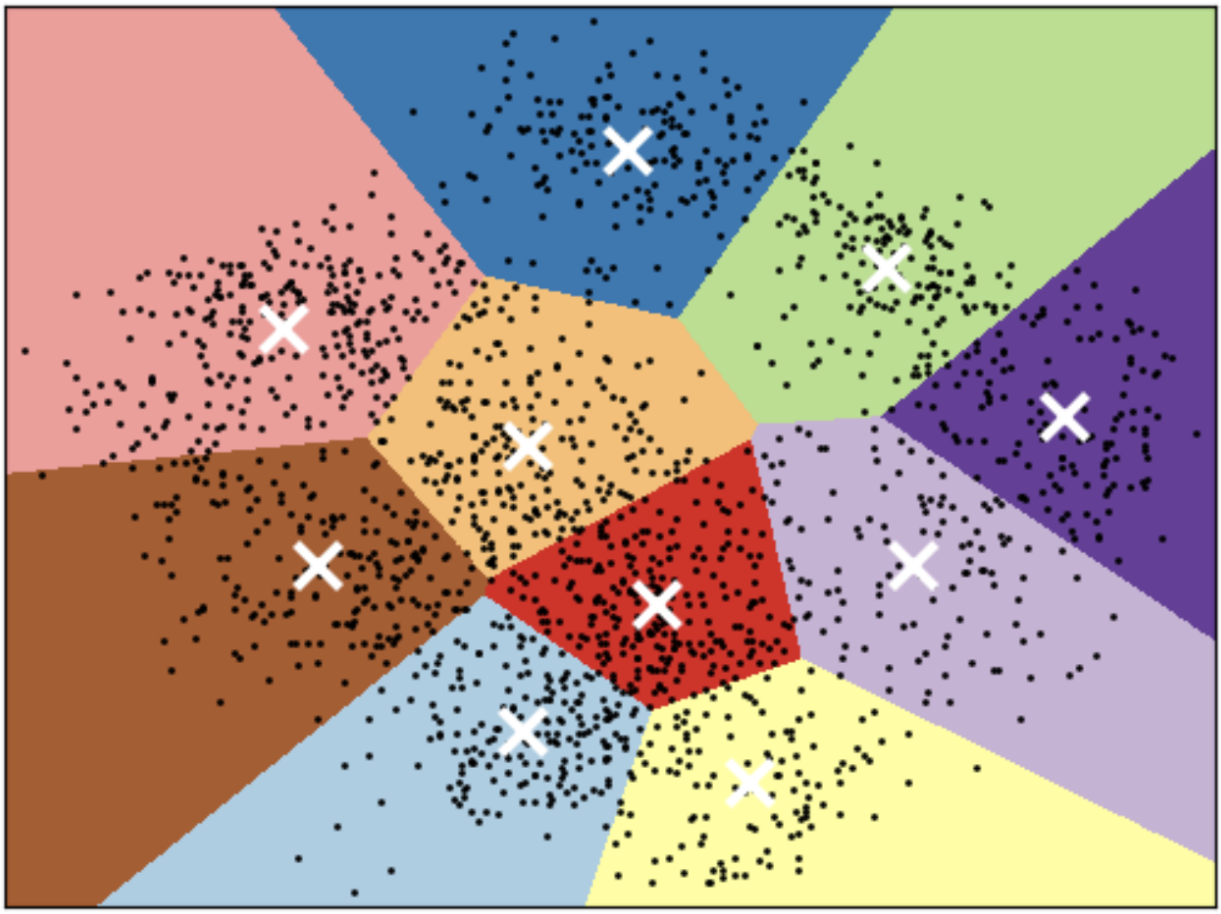
K-Means limitations

- Relies on random initialization and so the outcome may change depending upon this initialization.
- K-Means clustering requires to specify the number of clusters in advance.
- Very often you do not know the centers in advance. The elbow method or the silhouette method to find the optimal number of clusters are not always easy to interpret.
- Each point has to have a cluster assignment.

K-Means limitations: Shape of K-Means clusters

- K-Means partitions the space based on the closest mean.

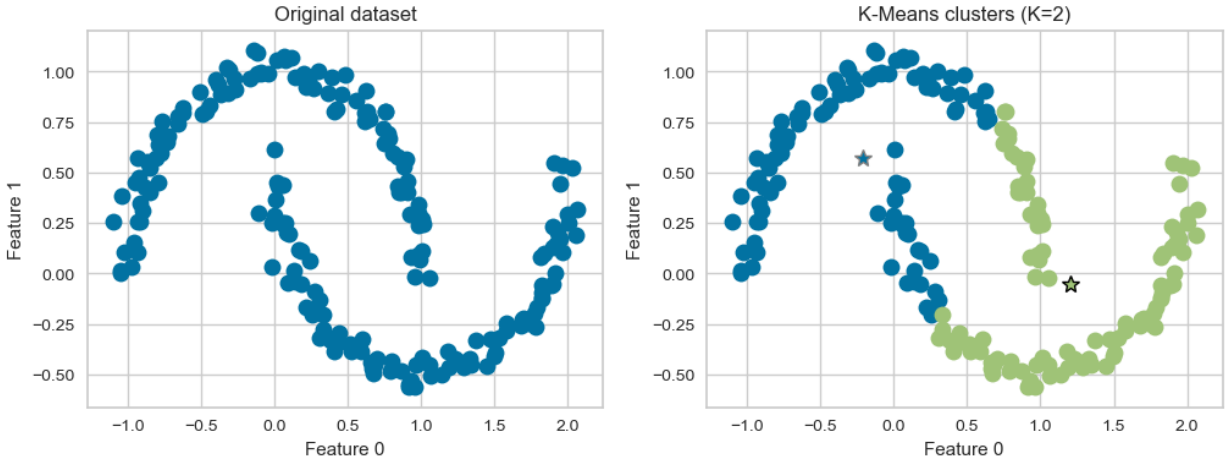
- Each cluster is defined solely by its center and so it can only capture relatively simple shapes.
- So the boundaries between clusters are linear; It fails to identify clusters with complex shapes. Source (https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html).



K-Means: failure case 1

- K-Means performs poorly if the clusters have more complex shapes (e.g., two moons data below).

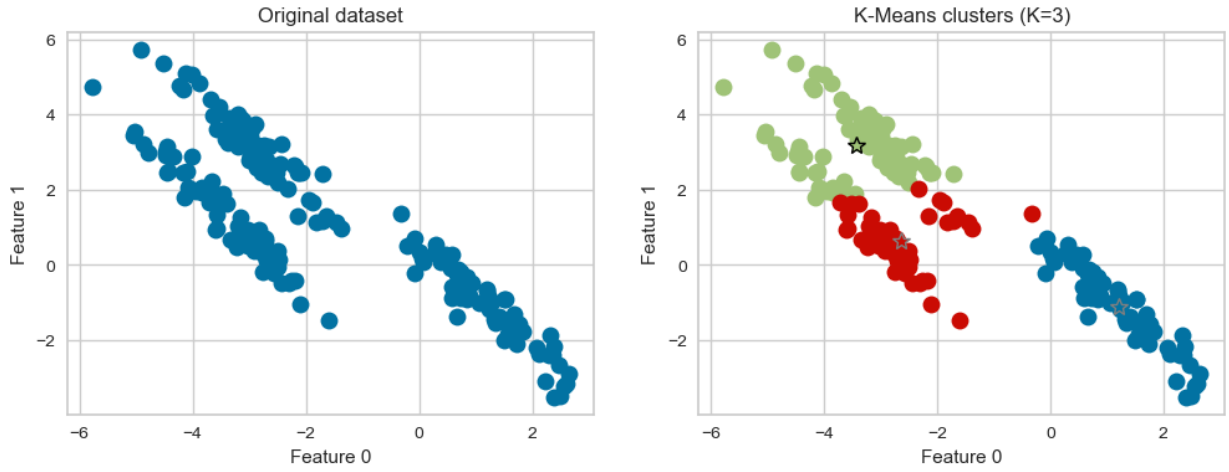
In [2]:
```
X, y = make_moons(n_samples=200, noise=0.05, random_state=42)
plot_X_k_means(X, k=2)
```



K-Means: failure case 2

- It assumes that all directions are equally important for each cluster and fails to identify non-spherical clusters.

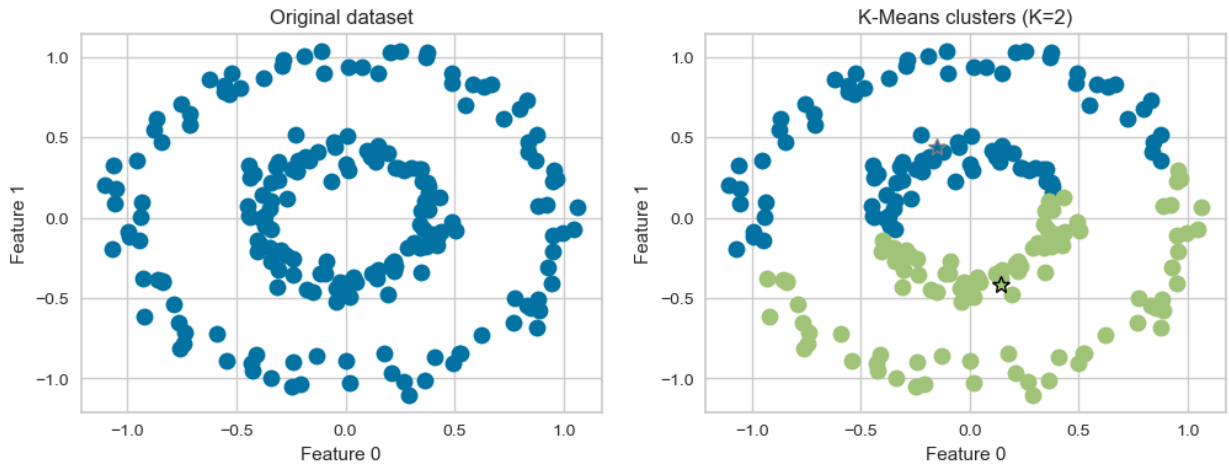
In [3]:
```
# generate some random cluster data
X, y = make_blobs(random_state=170, n_samples=200)
rng = np.random.RandomState(74)
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)
plot_X_k_means(X, 3)
```



K-Means: failure case 3

- Again, K-Means is unable to capture complex cluster shapes.

In [4]:
```
X = datasets.make_circles(n_samples=200, noise=0.06, factor=0.4)[0]
plot_X_k_means(X, 2)
```



- Can we do better than this?

DBSCAN

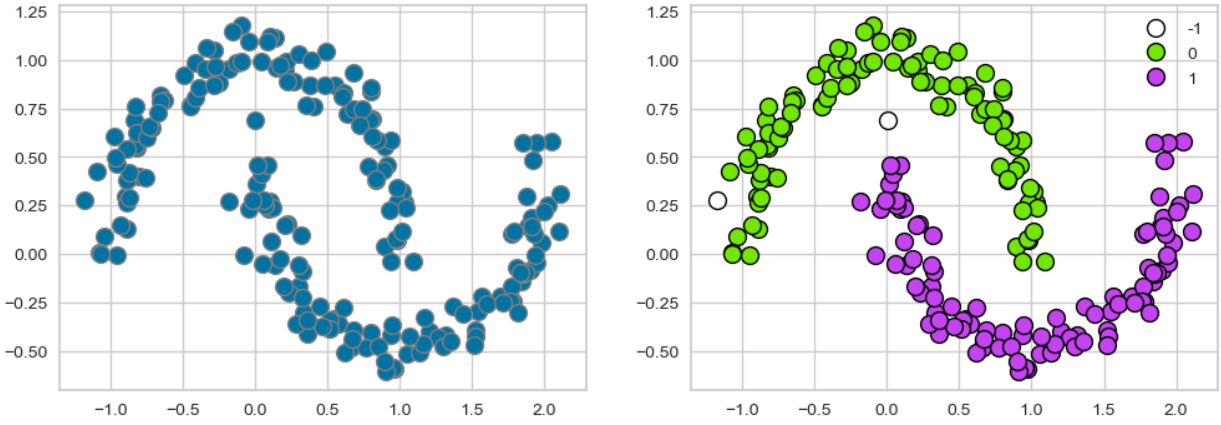
- **D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise

DBSCAN introduction

- DBSCAN is a density-based clustering algorithm.
- Intuitively, it's based on the idea that clusters form dense regions in the data and so it works by identifying "crowded" regions in the feature space.
- It can address some of the limitations of K-Means we saw above.
  - It does not require the user to specify the number of clusters in advance.
  - It can identify points that are not part of any clusters.
  - It can capture clusters of complex shapes.

Let's try `sklearn` 's DBSCAN.

In [5]:
```
X, y = make_moons(n_samples=200, noise=0.08, random_state=42)
dbscan = DBSCAN(eps=0.2)
dbscan.fit(X)
plot_X_dbscan(X, dbscan)
```



In [6]:
```
dbscan.labels_
```

Out[6]:
```
array([ 0,  0,  0,  1,  1,  1,  0,  1,  1,  0,  0,  1,  1,  0,  1,  1,
0,
        1,  0,  0,  1,  0,  1,  0,  0,  0,  0,  0,  1,  1,  0,  0,  0,
0,
        1,  1,  1,  1,  1,  0,  0,  1,  0,  0,  1,  1,  0,  0,  1,  0,
1,
        1,  0,  1,  1,  0,  1,  0,  1,  0,  1,  0,  0,  0,  0,  1,  0,
1,
        0,  1,  0,  1,  1,  0,  1,  1,  0,  1, -1,  1,  0,  0,  0,  1,
1,
        0,  1,  0,  1,  0,  0,  1,  1,  0,  0,  1,  1,  0,  0,  1,  0,
0,
        1,  1,  1,  1,  0,  0,  1,  1,  1,  0,  1,  1,  0,  1,  0,  0,
1,
        0,  0,  1,  1,  1,  0,  0,  1,  1,  0,  0,  1,  0,  0,  1,  0,
1,
        0,  1,  0,  1,  1,  1,  1,  0,  0,  0,  0,  1,  0,  0,  1,  1,
0,
        0,  0,  0,  1,  1,  0,  0,  1,  1,  1,  1,  0,  0,  1,  0,  1,
0,
        0,  0,  0,  0,  1,  1,  1,  0,  1,  1,  0,  0,  1,  1,  1,  1,
0,
        1,  0,  0,  1,  1,  0,  1,  0,  1,  0, -1,  1,  1])
```

- DBSCAN is able to capture half moons shape
- We don't not have to specify the number of clusters.
  - That said, it has two other non-trivial hyperparameters to tune.
- There are two examples which have not been assigned any label (noise examples).

One more example of DBSCAN clusters capturing complex cluster shapes.

In [71]:
```
X = datasets.make_circles(n_samples=200, noise=0.06, factor=0.4)[0]
dbscan = DBSCAN(eps=0.2, min_samples=3)
dbscan.fit(X)
plot_X_dbscan(X, dbscan)
```

How does it work?

- Iterative algorithm.
- Based on the idea that clusters form dense regions in the data.

Source (https://dashee87.github.io/data%20science/general/Clustering-with-Scikit-with-GIFs/)

Two main hyperparameters

- `eps` : determines what it means for points to be "close"

- `min_samples` : determines the number of **neighboring points** we require to consider in

Effect of `eps` hyperparameter

- `eps` : determines what it means for points to be "close"

In [8]:
```
X, y = make_blobs(random_state=0, n_samples=11)
```

In [72]:
```
# interactive(lambda eps=1: plot_dbscan_with_labels(X, eps), eps=(1, 12
plot_dbscan_with_labels(X, eps=1)
```

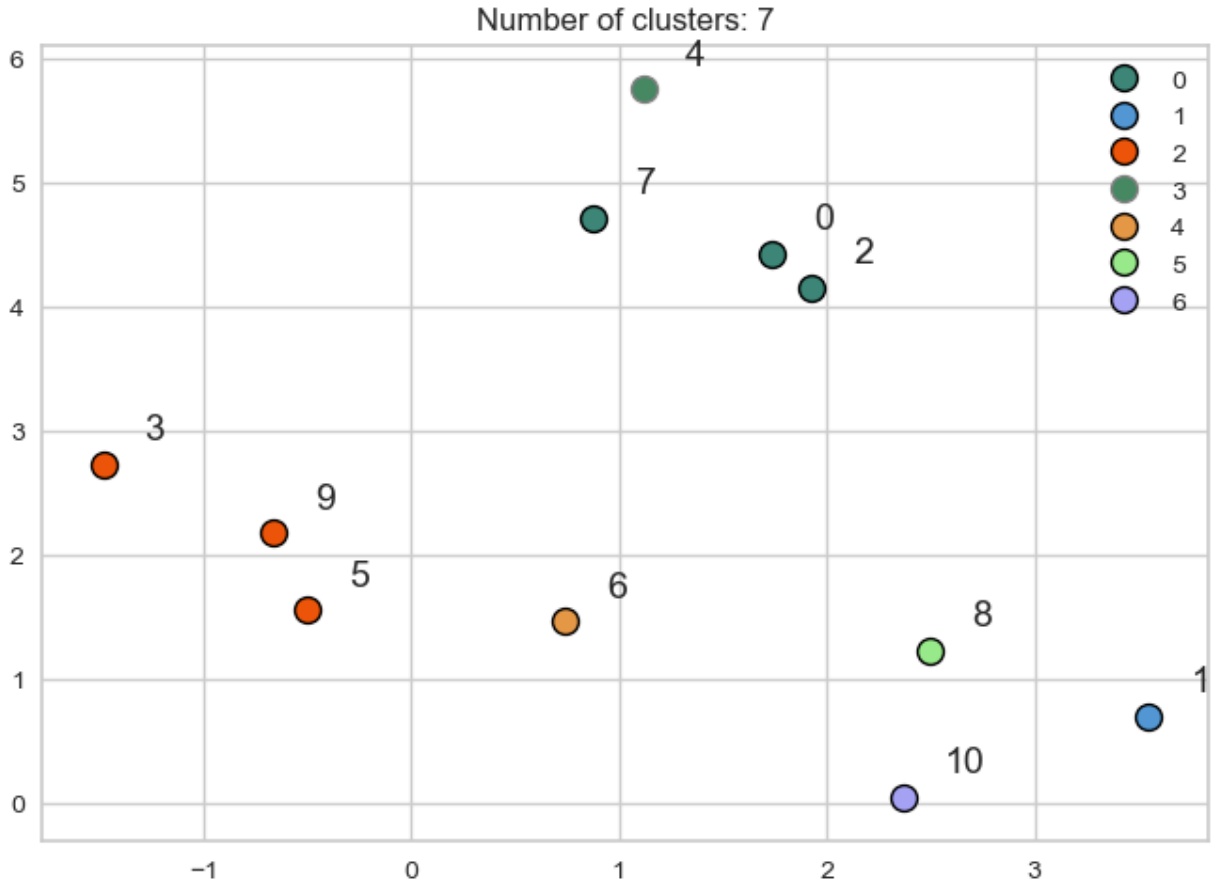
In [10]:
```
pd.DataFrame(euclidean_distances(X, X))
```

Out[10]:

	0	1	2	3	4	5	6	7	8	
0	0.000000	4.149512	0.332095	3.637398	1.468503	3.640916	3.124431	0.911272	3.283718	3.289
1	4.149512	0.000000	3.821671	5.427714	5.617833	4.137181	2.910477	4.830950	1.180379	4.465
2	0.332095	3.821671	0.000000	3.692209	1.796596	3.555578	2.937847	1.193839	2.976718	3.257
3	3.637398	5.427714	3.692209	0.000000	3.991555	1.535625	2.559670	3.078516	4.253314	0.989
4	1.468503	5.617833	1.796596	3.991555	0.000000	4.507219	4.311101	1.072565	4.732056	4.001
5	3.640916	4.137181	3.555578	1.535625	4.507219	0.000000	1.243674	3.447353	3.013883	0.645
6	3.124431	2.910477	2.937847	2.559670	4.311101	1.243674	0.000000	3.253476	1.771561	1.575
7	0.911272	4.830950	1.193839	3.078516	1.072565	3.447353	3.253476	0.000000	3.843925	2.966
8	3.283718	1.180379	2.976718	4.253314	4.732056	3.013883	1.771561	3.843925	0.000000	3.299
9	3.289419	4.465344	3.257210	0.989059	4.001437	0.645802	1.575484	2.966927	3.299630	0.000
10	4.427098	1.347556	4.132565	4.694590	5.849187	3.238439	2.158345	4.904326	1.194950	3.705

Effect of `min_samples` hyperparameter

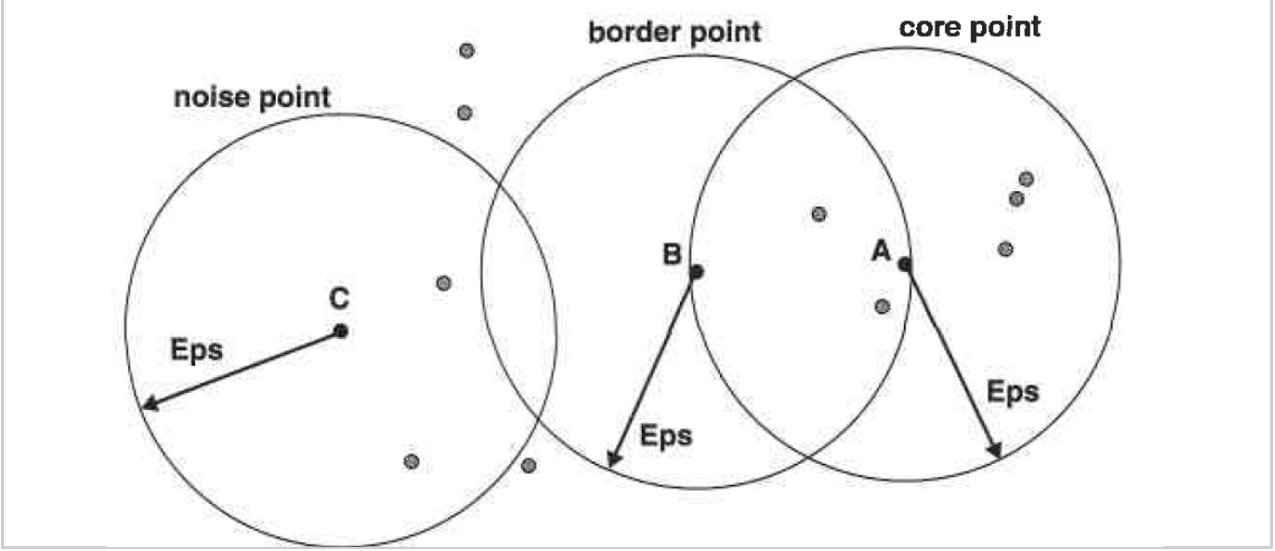
In [11]:
```
# interactive(lambda min_samples=1: plot_dbscan_with_labels(X, eps=1.0,
plot_dbscan_with_labels(X, eps=1.0, min_samples=1)
```



More details on DBSCAN

There are three kinds of points.

- **Core points** are the points that have at least `min_samples` points in the neighborhood.
- **Border points** are the points with fewer than `min_samples` points in the neighborhood, but are connected to a core point.
- **Noise points** are the points which do not belong to any cluster. In other words, the points which have less that `min_samples` point within distance `eps` of the starting point are noise points.



DBSCAN algorithm

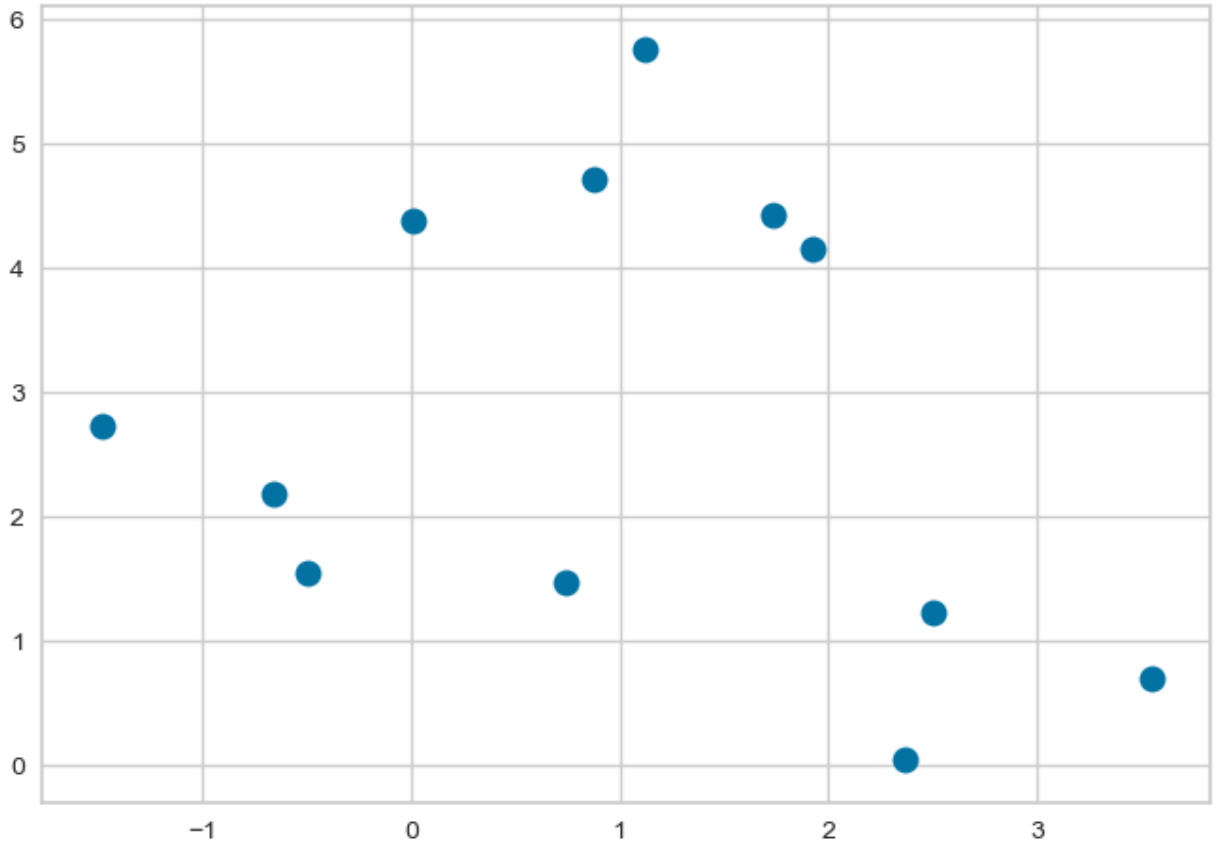
- Pick a point p at random.
- Check whether p is a "core" point or not. You can check this by looking at the number of neighbours within epsilon distance if they have at least `min_samples` points in the neighbourhood
- If p is a core point, give it a colour (label).
- Spread the colour of p to all of its neighbours.
- Check if any of the neighbours that received the colour is a core point, if yes, spread the colour to its neighbors as well.
- Once there are no more core points left to spread the colour, pick a new unlabeled point p and repeat the process.

K-Means vs. DBSCAN

- In DBSCAN, you do not have to specify the number of clusters!
  - Instead, you have to tune `eps` and `min_samples`.
- Unlike K-Means, DBSCAN doesn't have to assign all points to clusters.
  - The label is -1 if a point is unassigned.
- Unlike K-Means, there is no `predict` method.
  - DBSCAN only really clusters the points you have, not "new" or "test" points.

Illustration of hyperparameters `eps` and `min_samples`

In [12]:
```
X, y = make_blobs(random_state=0, n_samples=12)
mglearn.discrete_scatter(X[:, 0], X[:, 1]);
```



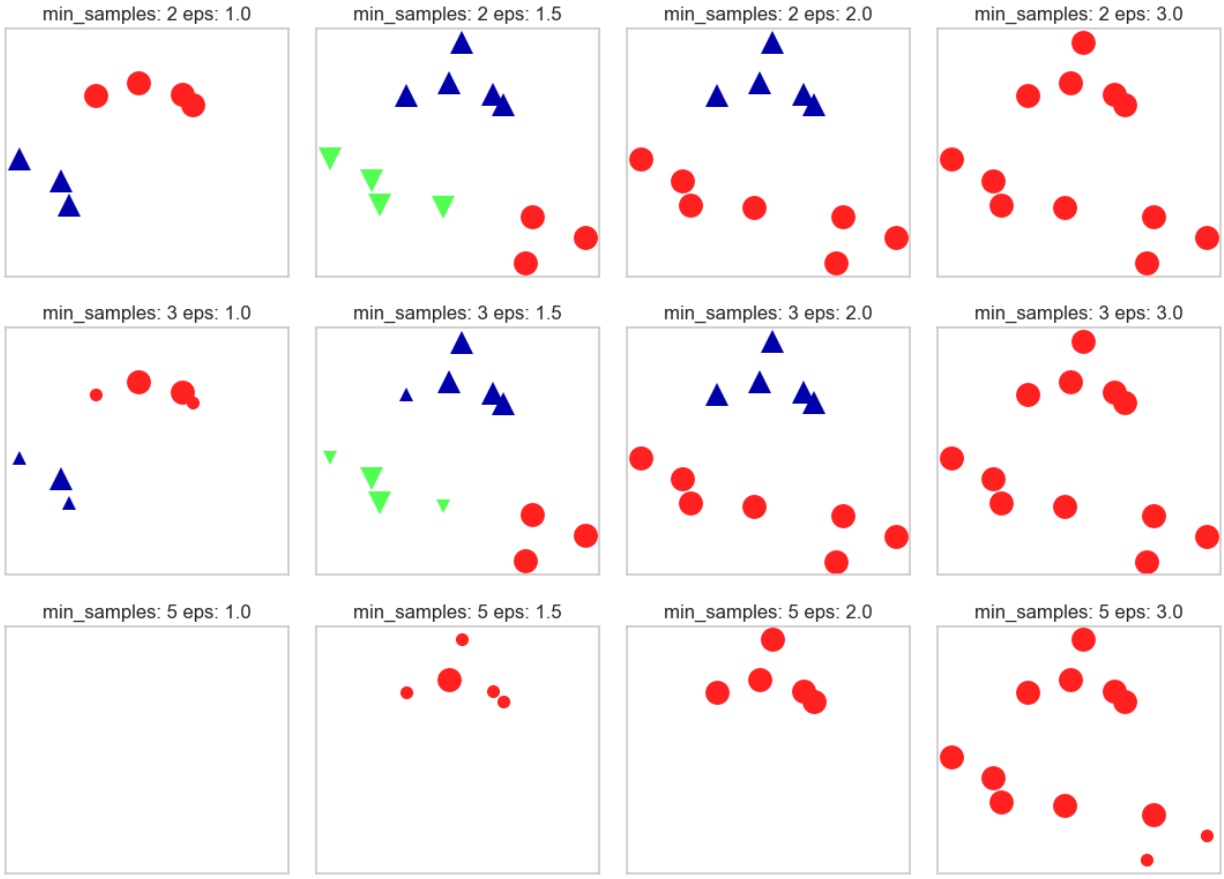
In [13]:
```
dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
print("Cluster memberships:{}".format(clusters))
```

```
Cluster memberships:[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

- Default values for hyperparameters don't work well on toy datasets.
- All points have been marked as noise with the default values for `eps` and `min_samples`
- Let's examine the effect of changing these hyperparameters.
  - noise points: shown in white
  - core points: bigger
  - border points: smaller

In [14]:
```
1 mglearn.plots.plot_dbscan()
```

```
min_samples: 2 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1
-1]
min_samples: 2 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 2 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 2 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 3 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1
-1]
min_samples: 3 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 3 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 3 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 5 eps: 1.000000  cluster: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1]
min_samples: 5 eps: 1.500000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1
-1]
min_samples: 5 eps: 2.000000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1
-1]
min_samples: 5 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
```



Observations

- Increasing `eps` ($\uparrow$) (left to right in the plot above) means more points will be included in a cluster.
  - `eps` = 1.0 either creates more clusters or more noise points, whereas eps=3.0 puts all points in one cluster with no noise points.
- Increasing `min_samples` ($\uparrow$) (top to bottom in the plot above) means points in less dense regions will either be labeled as their own cluster or noise.

  - `min_samples=2` , for instance, has none or only a fewer noise points whereas `min_samples=5` has several noise points.
- Here `min_samples` = 2.0 or 3.0 and `eps` = 1.5 is giving us the best results.
- In general, it's not trivial to tune these hyperparameters.

Question for you

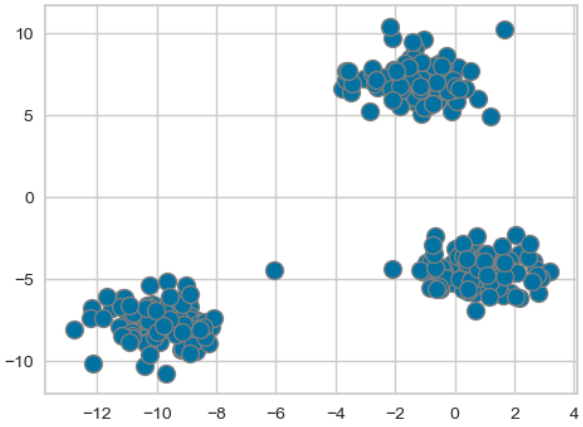
- Does the order that you pick the points matter in DBSCAN?

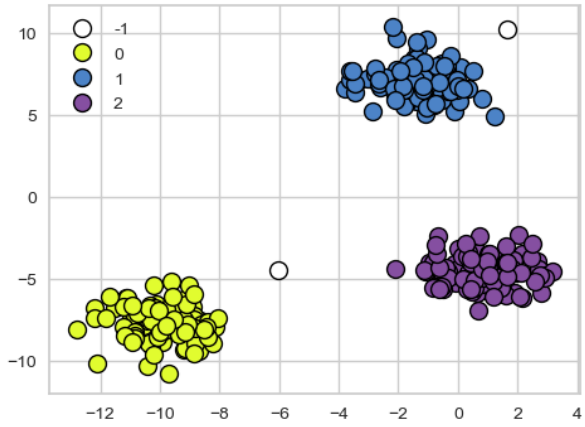
> No. Any of the cluster's core points is able to fully identify the cluster, with no randomness involved. The only possible conflict you might get is that if two clusters have the same border point. In this case the assignment will be implementation dependent, but usually the border point will be assigned to the first cluster that "finds" it.

Evaluating DBSCAN clusters

- We cannot use the elbow method to examine the goodness of clusters created with DBSCAN.
- But we can use the silhouette method because it's not dependent on the idea of cluster centers.

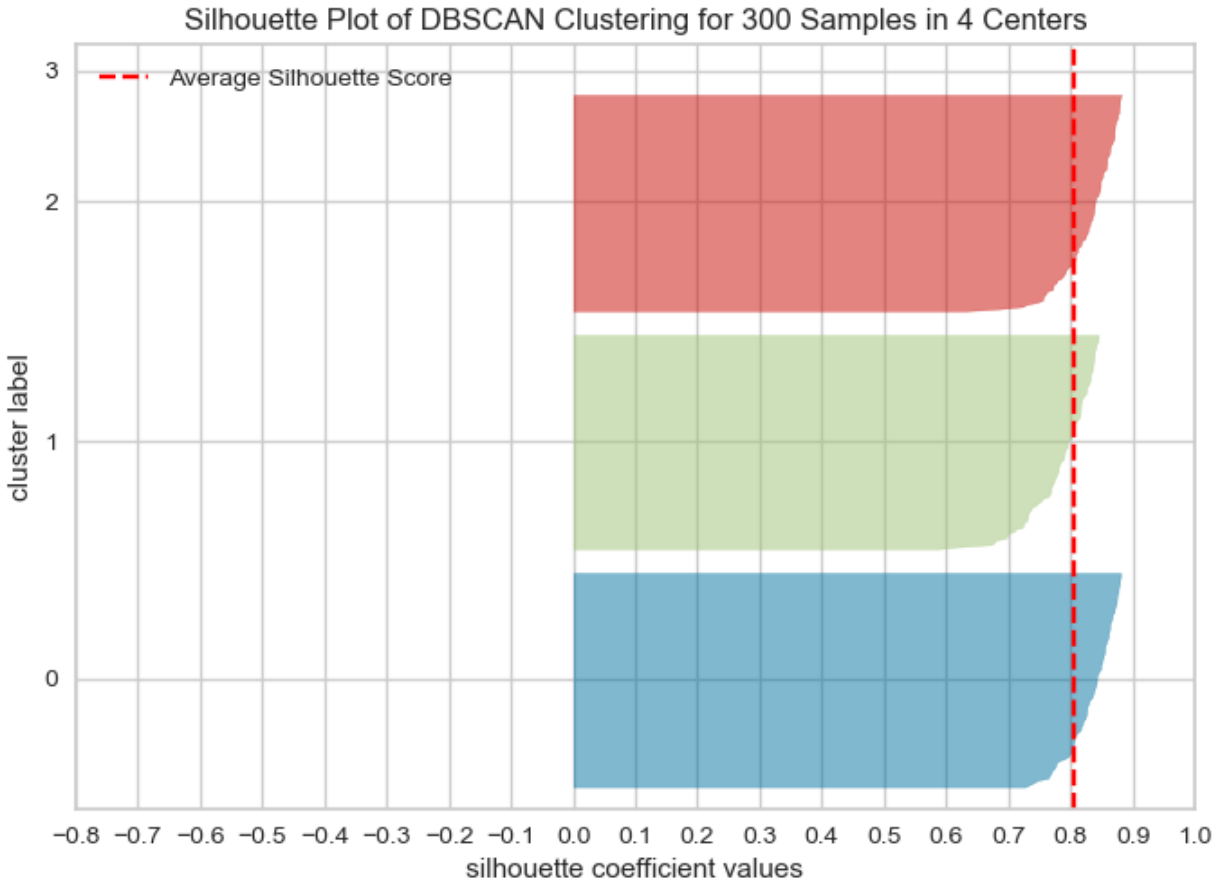
In [15]:
```
X, y = make_blobs(random_state=100, centers=3, n_samples=300)
dbscan = DBSCAN(eps=2, min_samples=5)
dbscan.fit(X)
plot_X_dbscan(X, dbscan)
```





In [16]:
```
# Yellowbrick is designed to work with K-Means and not with DBSCAN.
# So it needs the number of clusters stored in n_clusters
# It also needs `predict` method to be implemented.
# So I'm implementing it here so that we can use Yellowbrick to show Si
n_clusters = len(set(dbscan.labels_))
dbscan.n_clusters = n_clusters
dbscan.predict = lambda x: dbscan.labels_
```

In [17]:
```
visualizer = SilhouetteVisualizer(dbscan, colors="yellowbrick")
visualizer.fit(X)  # Fit the data to the visualizer
visualizer.show();
```



Summary: Pros and cons

- Pros
  - Can learn arbitrary cluster shapes
  - Can detect outliers
- Cons
  - Cannot `predict` on new examples.
  - Needs tuning of two non-obvious hyperparameters

There is an improved version of DBSCAN called `HDBSCAN` (hierarchical DBSCAN) (https://github.com/scikit-learn-contrib/hdbscan).

DBSCAN: failure cases

- DBSCAN is able to capture complex clusters. But this doesn't mean that `DBSCAN` always works better. It has its own problems!
- DBSCAN doesn't do well when we have clusters with different densities.
  - You can play with the hyperparameters but it's not likely to help much.

DBSCAN: failure cases

- Let's consider this dataset with three clusters of varying densities.
- K-Means performs better compared to DBSCAN. But it has the benefit of knowing the value of K in advance.

In [18]:
```
X_varied, y_varied = make_blobs(
    n_samples=200, cluster_std=[1.0, 5.0, 1.0], random_state=10
)
plot_k_means_dbscan_comparison(X_varied)
```

Hierarchical clustering

Motivation

- Deciding how many clusters we want is a hard problem.
- Often, it's useful to get a complete picture of similarity between points in our data before picking the number of clusters.
- Hierarchical clustering is helpful in these scenarios.

Main idea

1. Start with each point in its own cluster.
2. Greedily merge most similar *clusters*.
3. Repeat Step 2 until you obtain only one cluster ($n-1$ times).

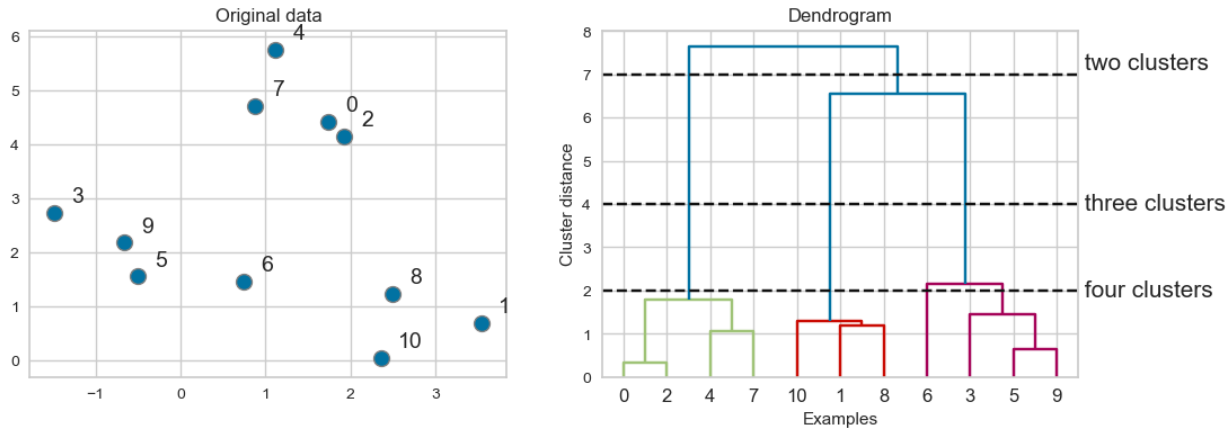
Visualizing hierarchical clustering

- Hierarchical clustering can be visualized using a tool called **a dendrogram**.
- Unfortunately, `sklearn` cannot do it so we will use the package `scipy.cluster.hierarchy` for hierarchical clustering.

Hierarchical clustering input and output

In [19]:
```
import scipy
X, y = make_blobs(random_state=0, n_samples=11)
linkage_array = scipy.cluster.hierarchy.ward(X)
```

In [20]:
```
plot_X_dendrogram(X, linkage_array, label_n_clusters=True)
```



- Every point goes through the journey of being on its own (its own cluster) and getting merged with some other bigger clusters.
- The intermediate steps in the process provide us clustering with different number of clusters.

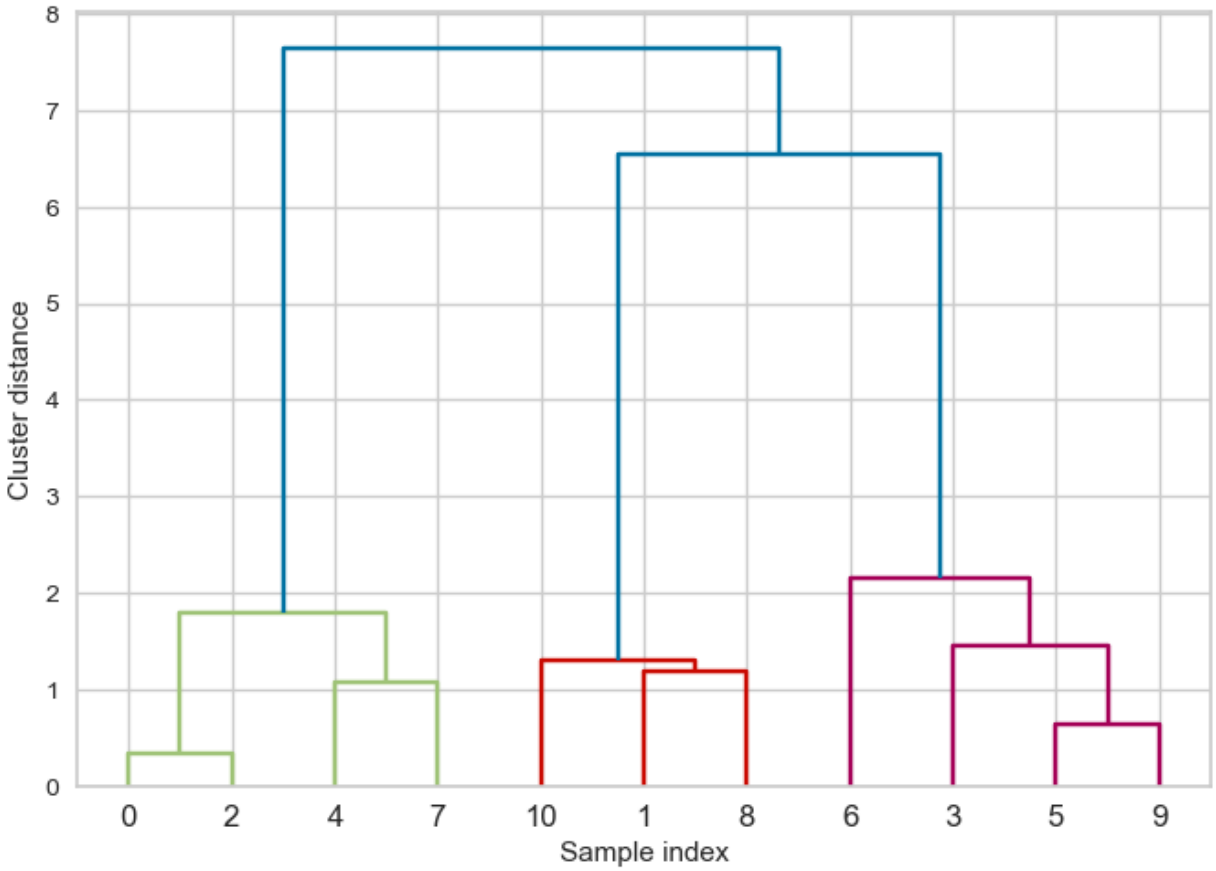
Dendrogram

- Dendrogram is a tree-like plot.
- On the x-axis we have data points.
- On the y-axis we have distances between clusters.
- We start with data points as leaves of the tree.
- New parent node is created for every two clusters that are joined.
- The length of each branch shows how far the merged clusters go.

- In the dendrogram above going from three clusters to two clusters means merging far apart points because the branches between three cluster to two clusters are long.

How to plot a dendrogram?

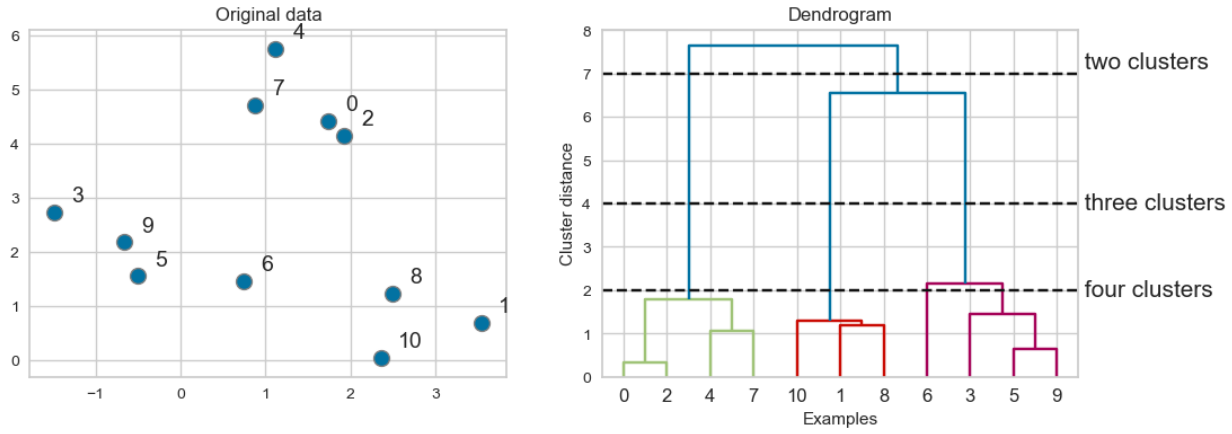
In [21]:
```
from scipy.cluster.hierarchy import dendrogram

ax = plt.gca()
dendrogram(linkage_array, ax=ax)
plt.xlabel("Sample index")
plt.ylabel("Cluster distance");
```



What do we mean by distance between clusters?

- We know how to measure distance between points (e.g., using Euclidean distance).
- How do we measure distances between clusters?
- The **linkage criteria** determines how to find similarity between clusters:
- Some example linkage criteria are:
  - single linkage (minimum distance)
  - average linkage (average distance)
  - complete (or maximum) linkage (maximum distance)
  - ward linkage

In [22]:
```
plot_X_dendrogram(X, linkage_array, label_n_clusters=True)
```



`single` linkage

- Merges two clusters that have the smallest minimum distance between all their points.
- Let's use `scipy.cluster.hierarchy`'s `single` to get linkage information.
- This method gives us matrix `Z` with the merging information.

In [23]:
```
from scipy.cluster.hierarchy import (
    average,
    complete,
    dendrogram,
    fcluster,
    single,
    ward,
)

Z = single(X)
columns = ["c1", "c2", "distance(c1, c2)", "# observations"]
```

In [24]:
```
pd.DataFrame(Z, columns=columns).head()
```

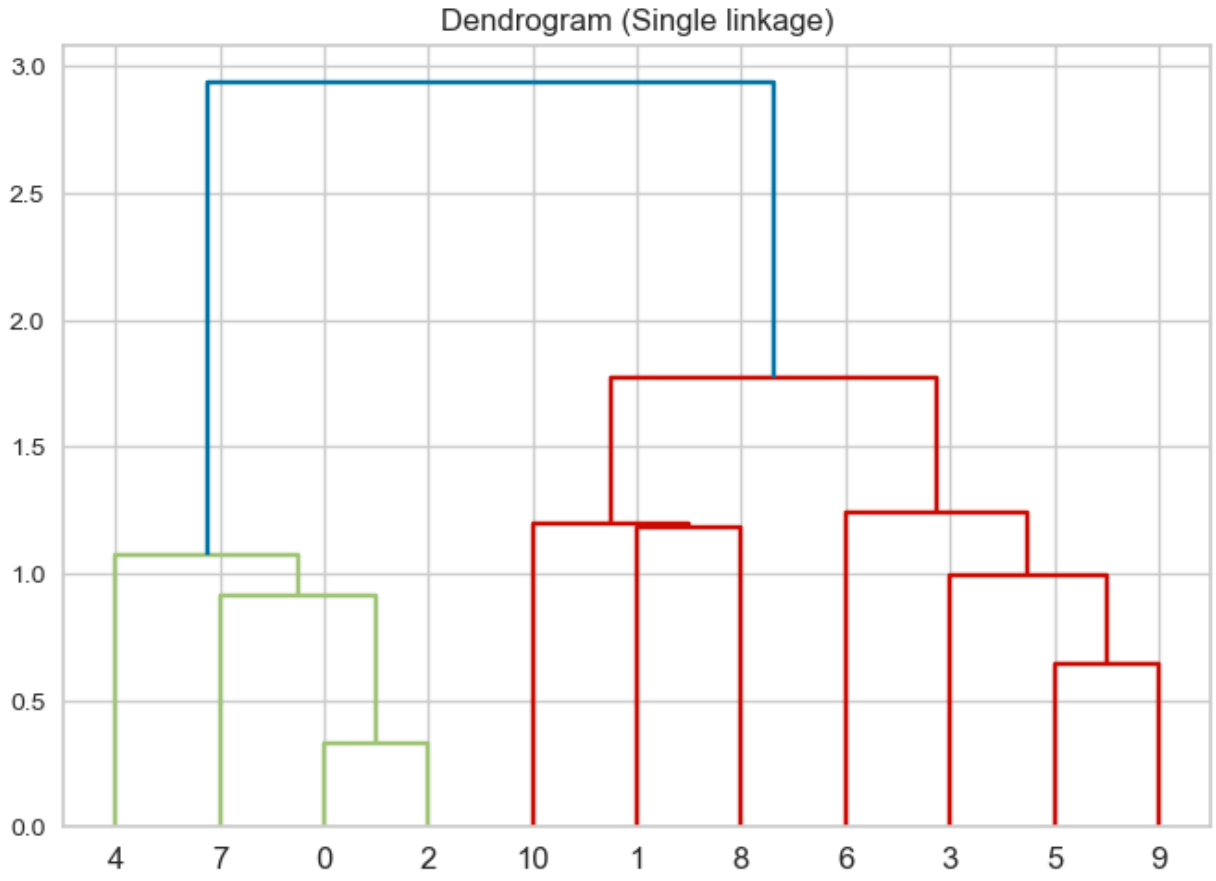
Out[24]:

	c1	c2	distance(c1, c2)	# observations
0	0.0	2.0	0.332095	2.0
1	5.0	9.0	0.645802	2.0
2	7.0	11.0	0.911272	3.0
3	3.0	12.0	0.989059	3.0
4	4.0	13.0	1.072565	4.0

- The linkage returns a matrix `Z` of shape `n-1` (number of iterations) by 4:
- The rows represent iterations.
- First and second columns (c1 and c2 above): indexes of the clusters being merged.
- Third column (distance(c1, c2)): the distance between the clusters being merged.
- Fourth column (# observations): the number of examples in the newly formed cluster.

Creating dendrogram with `single` linkage

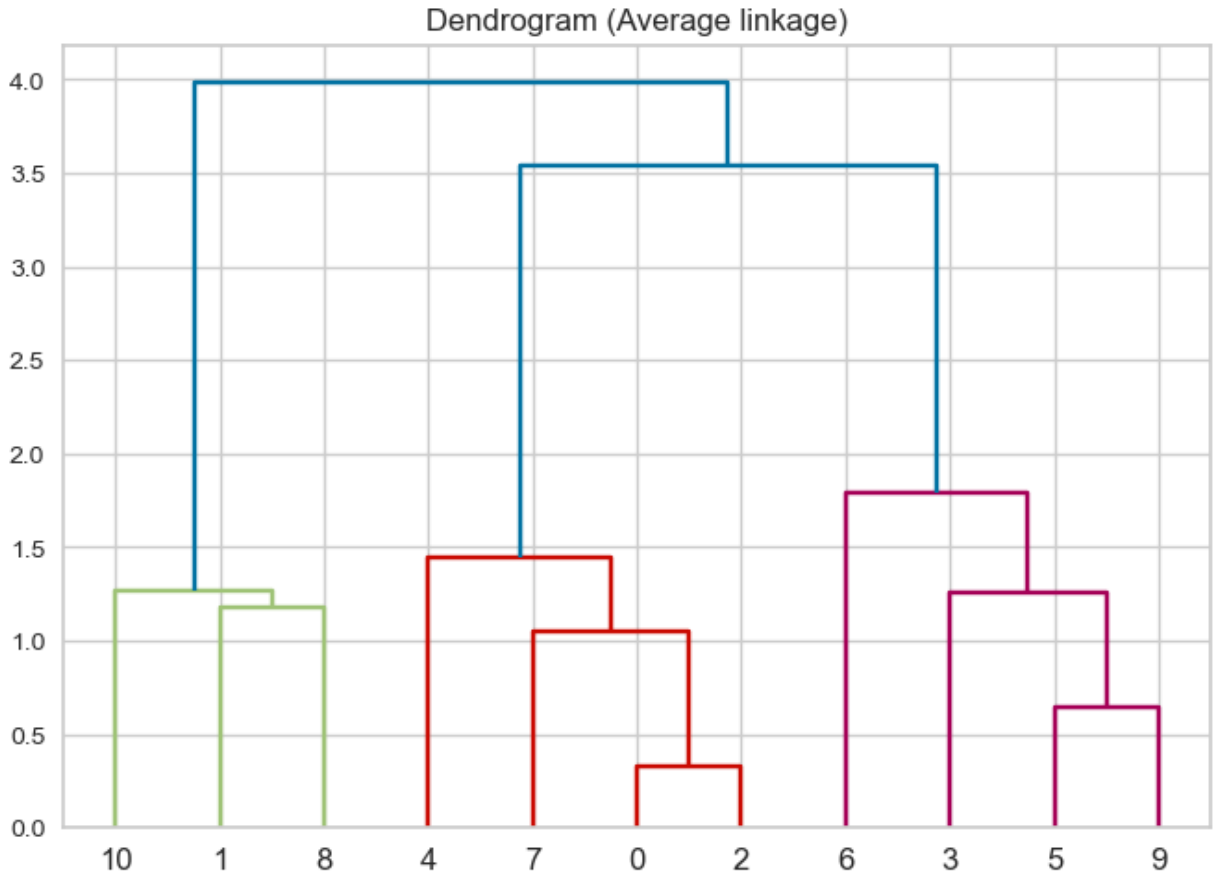
In [25]:
```
dendrogram(Z)
# Z is our single linkage matrix
plt.title("Dendrogram (Single linkage)");
```



`average` linkage

- Merges two clusters that have the smallest average distance between all their points.
- `scipy.cluster.hierarchy` 's `average` method gives us matrix `Z` with the merging information using average linkage.

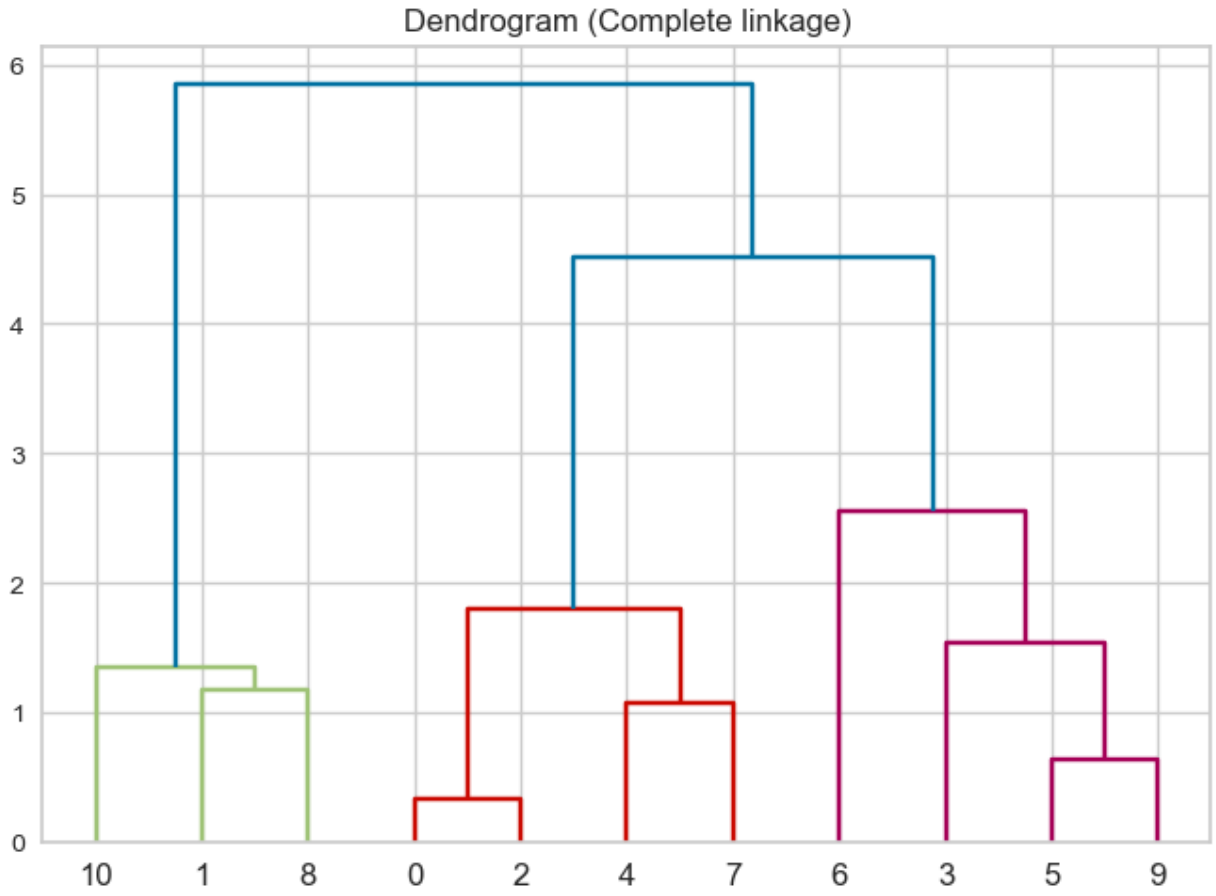
In [26]:
```
Z = average(X)
dendrogram(Z)
# Dendrogram with average linkage
plt.title("Dendrogram (Average linkage)");
```



`complete` linkage

- Merges two clusters that have the smallest maximum distance between their points.

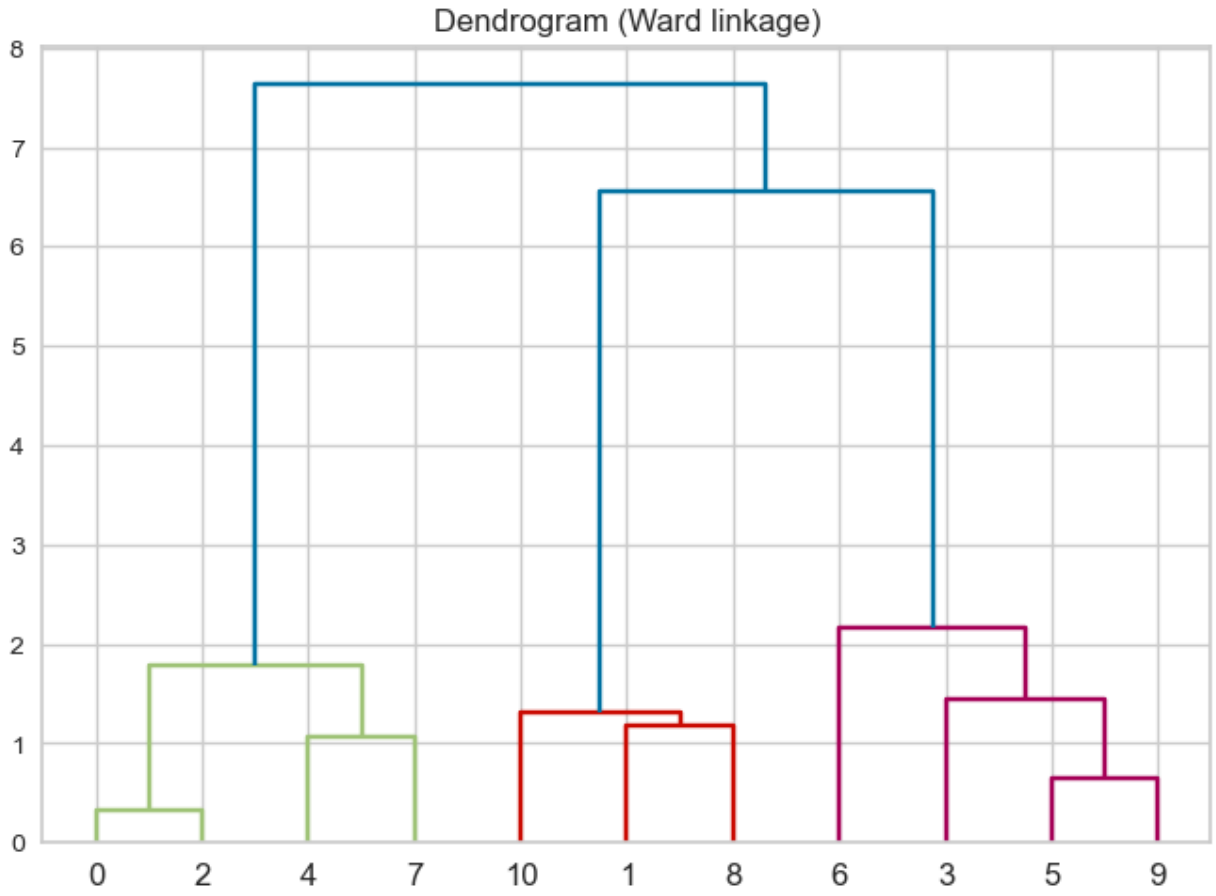
In [27]:
```
Z = complete(X)
dendrogram(Z)
# Dendrogram with complete linkage
plt.title("Dendrogram (Complete linkage)");
```



`ward` linkage

- Picks two clusters to merge such that the variance within all clusters increases the least.
- Often leads to equally sized clusters.

In [28]:
```
Z = ward(X)
dendrogram(Z)
# Dendrogram with ward linkage
plt.title("Dendrogram (Ward linkage)");
```



Hierarchical clustering on UN Subvotes dataset

- Let's use a dataset of votes on UN resolutions:

In [29]:
```
votes_df = pd.read_csv("../data/subvotes.csv")
votes = votes_df.pivot(index="country", columns="rcid")
votes = votes[np.sum(np.isnan(votes), axis=1) < 1]
print(votes.shape)
votes.head()
```

```
(17, 368)
```

Out[29]:

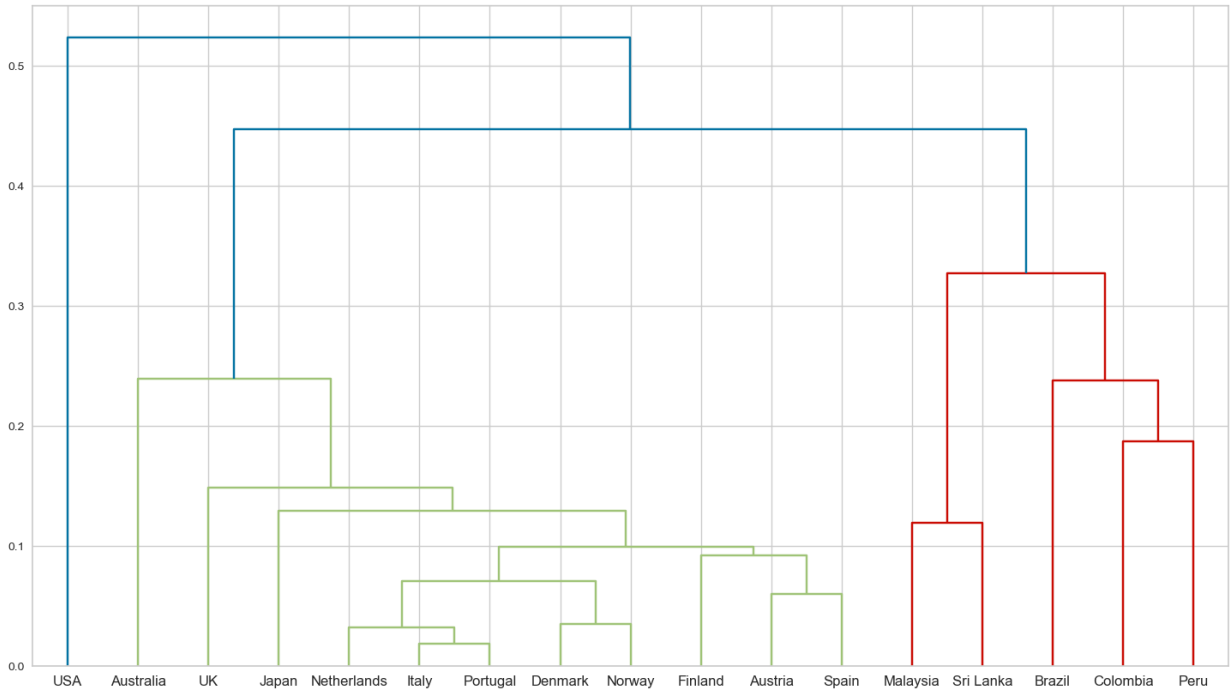
rcid	2491	2492	2497	2504	2510	2526	2563	2610	2641	2645	...	5321	5333	5337	5358
country															
Australia	1.0	2.0	1.0	3.0	3.0	3.0	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Austria	1.0	1.0	1.0	2.0	2.0	2.0	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Brazil	1.0	1.0	1.0	1.0	1.0	2.0	2.0	3.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Colombia	1.0	1.0	1.0	1.0	1.0	2.0	1.0	2.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Denmark	1.0	1.0	1.0	2.0	2.0	3.0	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0

5 rows × 368 columns

- We have 17 countries and 368 votes.
- Let's cluster countries based on how they vote.

- We'll use [hamming distance (https://en.wikipedia.org/wiki/Hamming_distance)](https://en.wikipedia.org/wiki/Hamming_distance) here because we are interested in knowing whether the countries agreed or disagreed on resolutions.

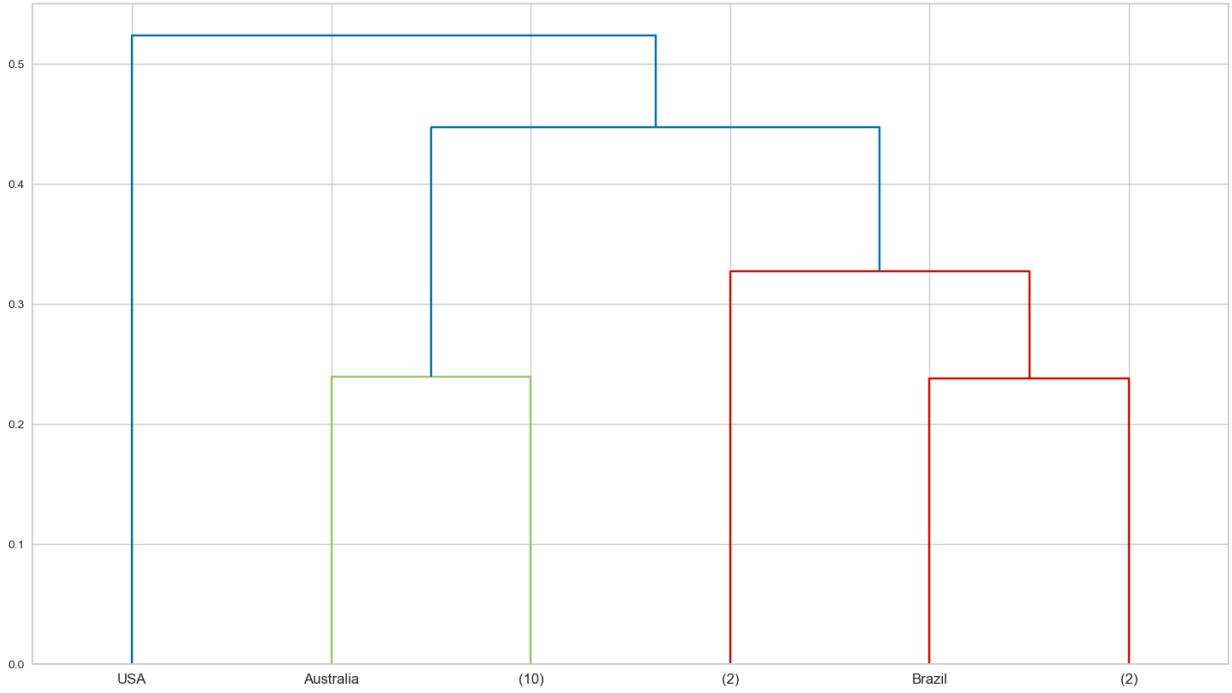
In [30]:
```
Z = linkage(votes, method="average", metric="hamming")
fig, ax = plt.subplots(figsize=(18, 10))
dendrogram(Z, p=6, ax=ax, labels=votes.index);
```



(Optional) Truncation

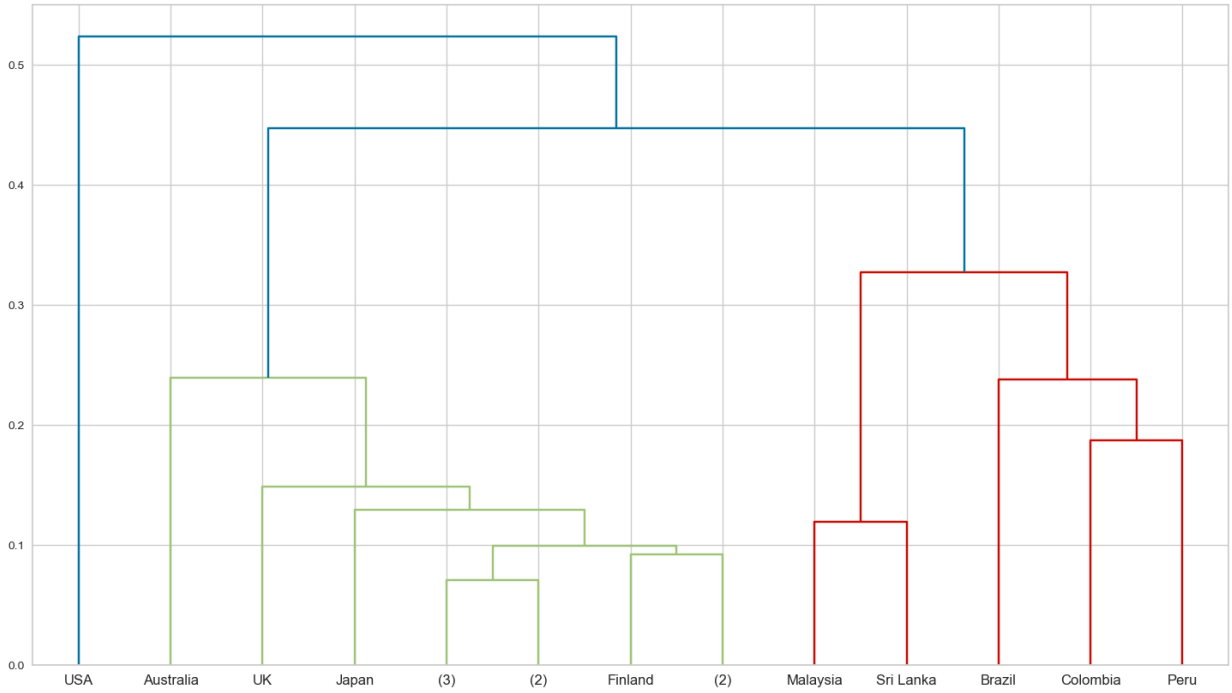
- If you want to truncate the tree in the dendrogram (specially when you have a big n) you can use the `truncate_mode` .

In [31]:
```
Z = linkage(votes, method="average", metric="hamming")
fig, ax = plt.subplots(figsize=(18, 10))
dendrogram(Z, p=6, truncate_mode="lastp", ax=ax, labels=votes.index);
# p is the number of leaves when truncate mode is "lastp"
```



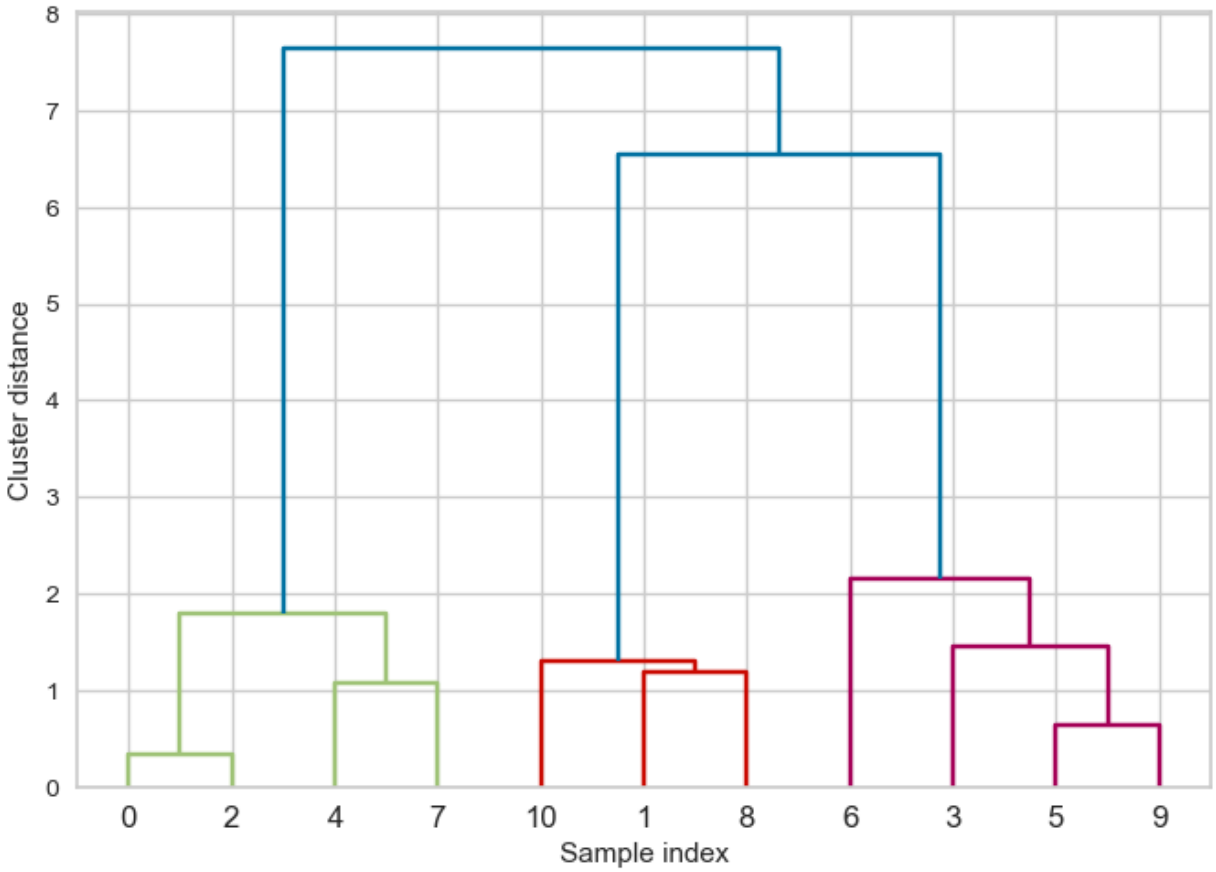
- Alternatively, you can truncate the tree down to x levels from the single cluster:

In [32]:
```
fig, ax = plt.subplots(figsize=(18, 10))
dendrogram(Z, p=6, truncate_mode="level", ax=ax, labels=votes.index);
# p is the max depth of the tree when truncate_mode is "level"
```

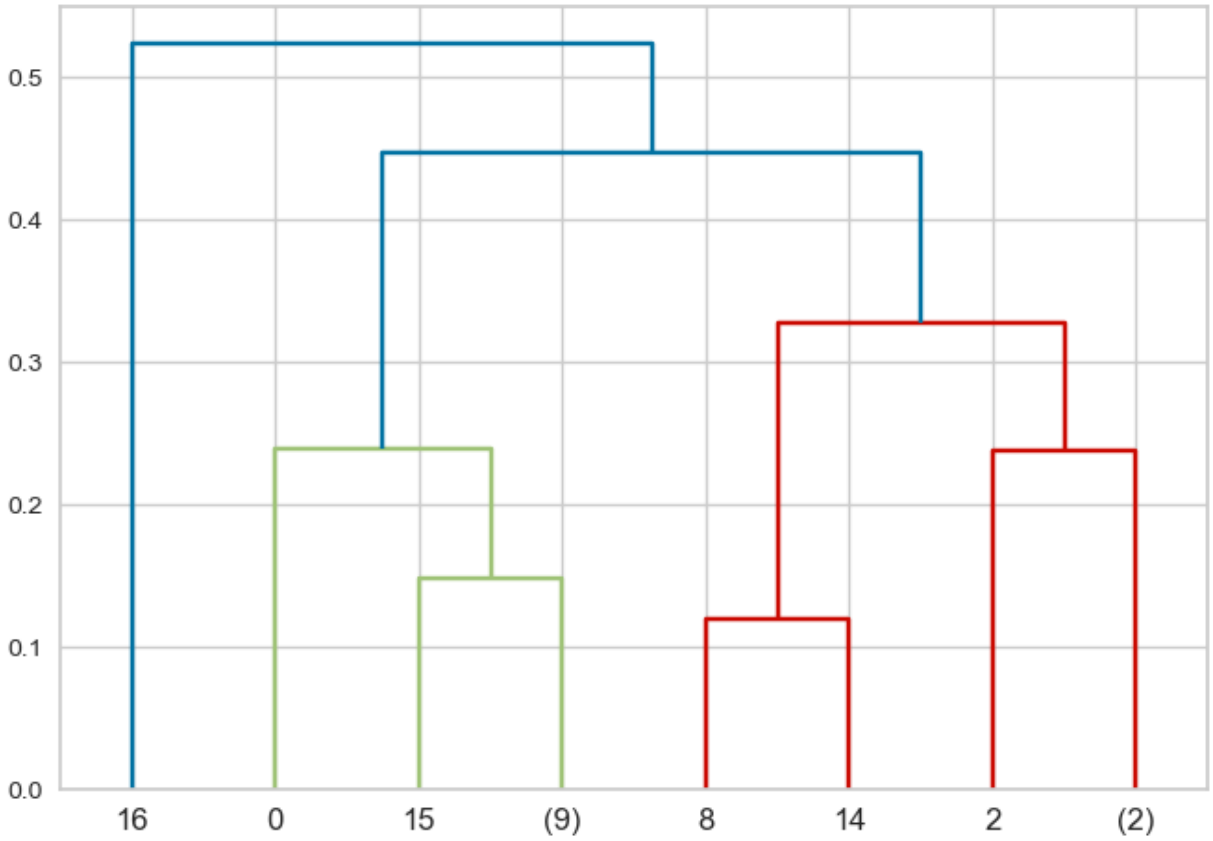


Let's go back to our toy dataset to understand truncation better.

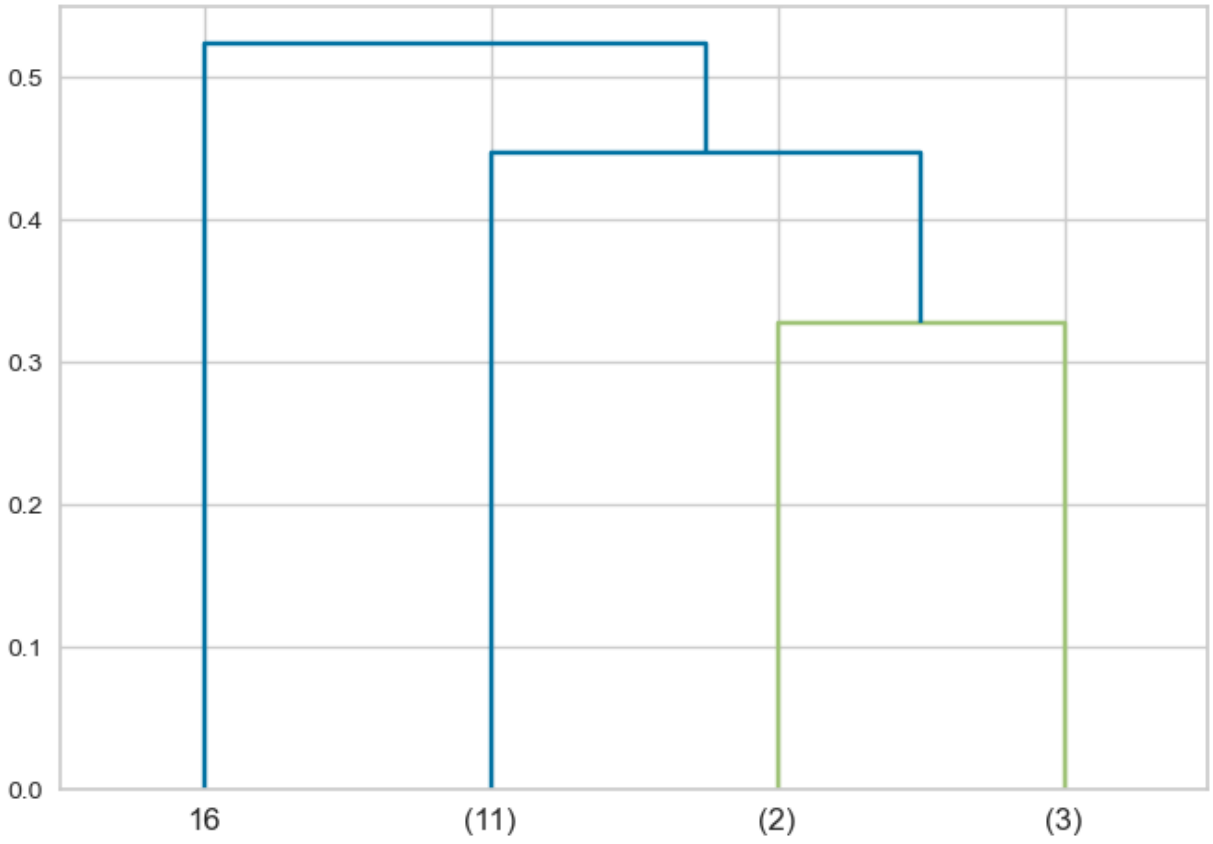
In [33]:
```
ax = plt.gca()
dendrogram(linkage_array, ax=ax)
plt.xlabel("Sample index")
plt.ylabel("Cluster distance");
```



In [34]:
```
dendrogram(Z, p=3, truncate_mode="level");
# p is the max depth of the tree
```



In [35]:
```
dendrogram(Z, p=4, truncate_mode="lastp");
# p is the number of leaf nodes
```



Flat cluster

- To bring the clustering to a "flat" format, we can use `fcluster`

In [36]:
```
from scipy.cluster.hierarchy import fcluster

cluster_labels = fcluster(Z, 6, criterion="maxclust")
```

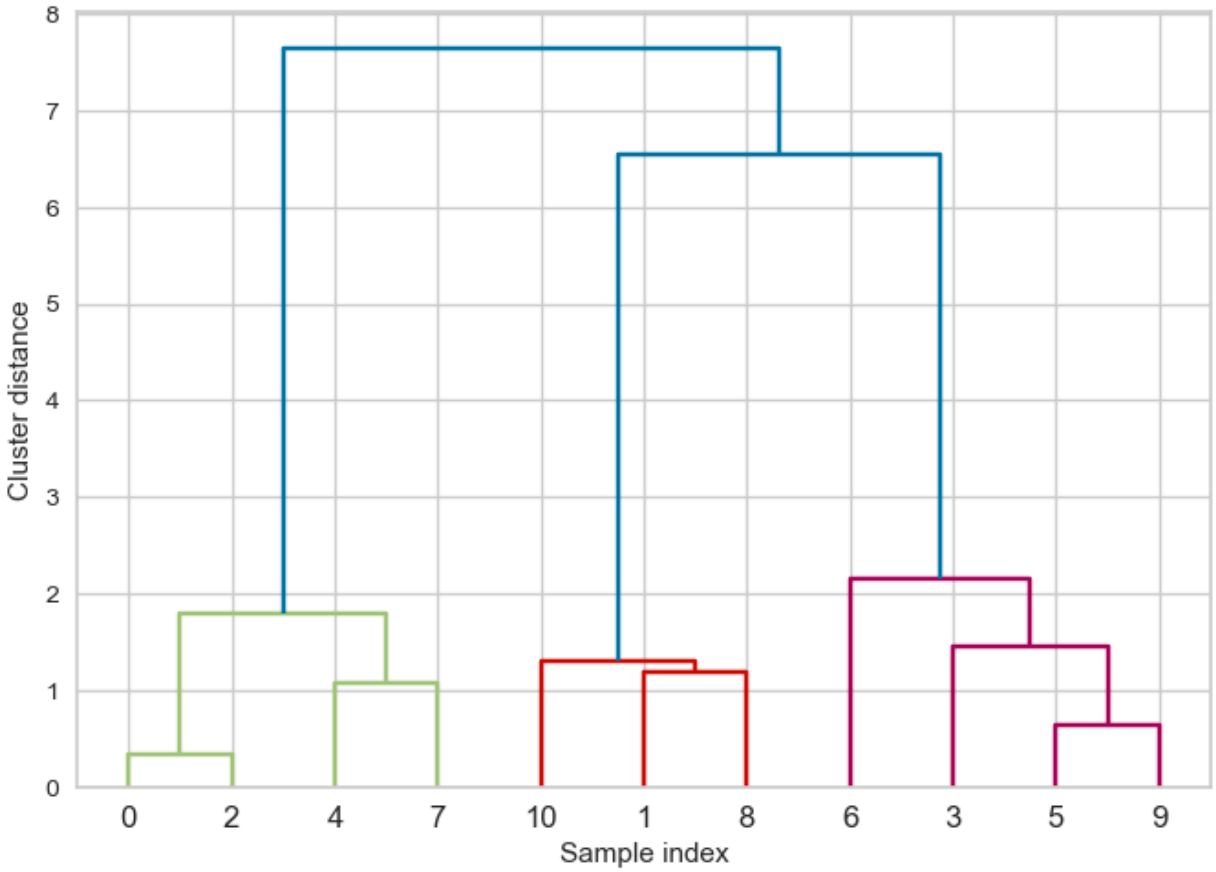
In [37]:
```
1 pd.DataFrame(cluster_labels, votes.index)
```

Out[37]:

country	0
Australia	2
Austria	1
Brazil	5
Colombia	4
Denmark	1
Finland	1
Italy	1
Japan	1
Malaysia	3
Netherlands	1
Norway	1
Peru	4
Portugal	1
Spain	1
Sri Lanka	3
UK	1
USA	6

To understand this better, let's try it out on our toy dataset.

In [38]:
```
ax = plt.gca()
dendrogram(linkage_array, ax=ax)
plt.xlabel("Sample index")
plt.ylabel("Cluster distance");
```



In [39]:
```
cluster_labels = fcluster(linkage_array, 3, criterion="maxclust")
pd.DataFrame(cluster_labels, columns=["Cluster"])
```

Out[39]:

	Cluster
0	1
1	2
2	1
3	3
4	1
5	3
6	3
7	1
8	2
9	3
10	2

? ? Questions for you

Which statements are TRUE?

- (A) With tiny epsilon (`eps` in `sklearn`) and min samples=1 (`min_samples=1` in `sklearn`) we are likely to end up with each point in its own cluster.
- (B) With a smaller value of `eps` and larger number for `min_samples` we are likely to end up with a one big cluster.
- (C) K-Means is more susceptible to outliers compared to DBSCAN.
- (D) In DBSCAN to be part of a cluster, each point must have at least `min_samples` neighbours in a given radius (including itself).
- (E) In DBSCAN, it is generally a good idea to run DBSCAN with a large number of different random orderings of training examples.

Which statements are TRUE?

- (A) In hierarchical clustering we do not have to worry about initialization.
- (B) Hierarchical clustering can only be applied to smaller datasets because dendrograms are hard to visualize for large datasets.
- (C) In all the three clustering methods we saw (K-Means, DBSCAN, hierarchical clustering), there is a way to decide the granularity of clustering (i.e., how many clusters to pick).
- (D) To get robust clustering we can naively ensemble cluster labels (e.g., pick the most popular label) produced by different clustering methods.
- (E) If you have a high Silhouette score and very clean and robust clusters, it means that the algorithm has captured the semantic meaning in the data of our interest.

Applying clustering on face images

- We'll be working with `sklearn` 's Labeled Faces in the Wild dataset (https://scikit-learn.org/0.16/datasets/labeled_faces.html).
- The dataset has images of celebrities from the early 2000s downloaded from the internet.

> Credit: This example is based on the example from here (https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/ch03.html).

In [40]:
```
import matplotlib as mpl
from sklearn.datasets import fetch_lfw_people

mpl.rcParams.update(mpl.rcParamsDefault)
plt.rcParams["image.cmap"] = "gray"
```

Example images from the dataset

In [41]:
```
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)

fig, axes = plt.subplots(2, 5, figsize=(10, 5), subplot_kw={"xticks": (
for target, image, ax in zip(people.target, people.images, axes.ravel()
    ax.imshow(image)
    ax.set_title(people.target_names[target])
```

In [42]:
```
plt.show()
```

In [43]:
```
image_shape = people.images[0].shape
print("people.images.shape: {}".format(people.images.shape))
print("Number of classes: {}".format(len(people.target_names)))
```

```
people.images.shape: (3023, 87, 65)
Number of classes: 62
```

There are 3,023 images stored as arrays of 5655 pixels (87 by 65), of 62 different people:

In [44]:
```
counts = np.bincount(people.target)  # count how often each target appe
df = pd.DataFrame(counts, columns=["count"], index=people.target_names)
df.sort_values("count", ascending=False)
```

Out[44]:

	count
George W Bush	530
Colin Powell	236
Tony Blair	144
Donald Rumsfeld	121
Gerhard Schroeder	109
...	...
Angelina Jolie	20
Jiang Zemin	20
Paul Bremer	20
Igor Ivanov	20
Michael Bloomberg	20

62 rows × 1 columns

Let's make the data less skewed by taking only 20 images of the each person.

In [45]:
```
mask = np.zeros(people.target.shape, dtype=bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:20]] = 1

X_people = people.data[mask]
y_people = people.target[mask]
```

In [46]:
```
X_people.shape
```

Out[46]: (1240, 5655)

Representation of images

- Representation of input is very important when you cluster examples.
- In this example, we'll use PCA representation. (We won't learn about PCA in this course. You can think of it as a method to extract most important information from the given data.)

In [47]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=100, whiten=True, random_state=0)
X_pca = pca.fit_transform(X_people)
```

Clustering faces with K-Means

We'll cluster the images with this new representation.

In [48]:
```
km = KMeans(n_clusters=10, random_state=10)
km.fit(X_pca)
```

Out[48]:
```
KMeans(n_clusters=10, random_state=10)
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

What are the sizes of the clusters?

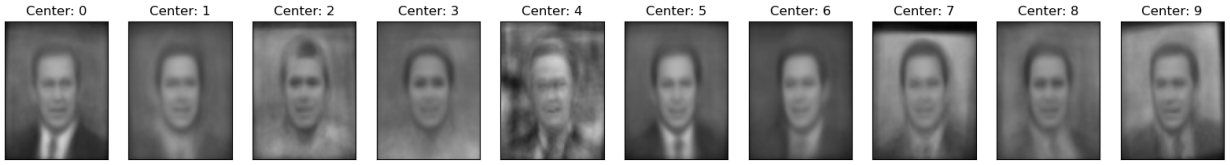
In [49]:
```
labels_km = km.fit_predict(X_pca)
print("Cluster sizes k-means: {}".format(np.bincount(labels_km)))
```

```
Cluster sizes k-means: [ 79 243  41 125   3 207 279  75 125  63]
```

Let's examine cluster centers. Are they going to be real images from the dataset?

In [50]:
```
def get_cluster_images(km, X_people, y_people, target_names, X_pca=None
    image_shape = (87, 65)
    fig, axes = plt.subplots(1, 6, subplot_kw={'xticks': (), 'yticks':
                             figsize=(10, 10), gridspec_kw={"hspace": .
    center = km.cluster_centers_[cluster]

    mask = km.labels_ == cluster

    if pca:
        dists = np.sum((X_pca - center) ** 2, axis=1)
        dists[~mask] = np.inf
        inds = np.argsort(dists)[:5]
        axes[0].imshow(pca.inverse_transform(center).reshape(image_shap
    else:
        dists = np.sum((X_people - center) ** 2, axis=1)
        dists[~mask] = np.inf
        inds = np.argsort(dists)[:5]
        axes[0].imshow(center.reshape(image_shape), vmin=0, vmax=1)

    axes[0].set_title('Cluster center %d'%(cluster))
    i = 1
    for image, label in zip(X_people[inds], y_people[inds]):
        axes[i].imshow(image.reshape(image_shape), vmin=0, vmax=1)
        axes[i].set_title("%s" % (target_names[label].split()[-1]), fon
        i+=1
    plt.show()
```

In [51]:
```
plot_faces_cluster_centers(km, pca)
```



- The centers found by K-Means are smooth versions of faces which makes sense.
- Intuitively, they seem to capture some interesting characteristics of faces:
  - grumpy faces
  - faces somewhat rotated to the left/right
  - Smiley faces

Let's examine images for different centers.

In [52]:
```
get_cluster_images(
    km, X_people, y_people, people.target_names, pca=pca, X_pca=X_pca,
)
```

In [53]:
```
get_cluster_images(
    km, X_people, y_people, people.target_names, pca=pca, X_pca=X_pca,
)
```

Clustering faces with DBSCAN

In [54]:
```
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

```
Unique labels: [-1]
```

With default hyperparameters, we get all points as noise points.

Tuning `eps`

In [55]:
```
people.target_names[y_people].shape
```

Out[55]:
```
(1240,)
```

Let's examine at distances between images.

In [56]:
```
from sklearn.metrics.pairwise import euclidean_distances

dists = euclidean_distances(X_pca)
np.fill_diagonal(dists, np.inf)

dist_df = pd.DataFrame(
    dists, index=people.target_names[y_people], columns=people.target_n
)

dist_df.iloc[10:20, 10:20]
```

Out[56]:

	George W Bush	George W Bush	Nestor Kirchner	Jean Chretien	Bill Clinton	George W Bush	Carlos Menem	Alvaro Uribe	G
George W Bush	inf	13.494180	12.842854	15.728106	15.821581	13.411802	15.145203	16.356001	1
George W Bush	13.494180	inf	11.172112	15.952421	16.520470	12.882513	14.535444	15.917523	1
Nestor Kirchner	12.842854	11.172112	inf	13.967069	14.954653	9.952094	12.282001	13.954332	
Jean Chretien	15.728106	15.952421	13.967069	inf	17.792767	15.888998	16.187300	17.894358	1
Bill Clinton	15.821581	16.520470	14.954653	17.792767	inf	15.382248	17.404568	16.727720	1
George W Bush	13.411802	12.882513	9.952094	15.888998	15.382248	inf	14.127582	15.053101	
Carlos Menem	15.145203	14.535444	12.282001	16.187300	17.404568	14.127582	inf	15.061512	1
Alvaro Uribe	16.356001	15.917523	13.954332	17.894358	16.727720	15.053101	15.061512	inf	1
George W Bush	11.605841	11.627776	8.903399	15.051611	14.883553	9.760360	11.510704	13.337687	
Colin Powell	12.597251	12.644590	12.063414	15.960009	15.612200	12.570139	14.349072	14.868257	1

In [57]:
```
1 dist_df.describe()
```

Out[57]:

	Winona Ryder	Jean Chretien	Carlos Menem	Ariel Sharon	Alvaro Uribe	Colin Powell	Recep Tayyip Erdogan
count	1240.000000	1240.000000	1240.000000	1240.000000	1240.000000	1240.000000	1240.000000
mean	inf	inf	inf	inf	inf	inf	inf
std	NaN	NaN	NaN	NaN	NaN	NaN	NaN
min	10.376113	8.781528	7.795739	11.128681	10.745722	8.586018	6.987148
25%	13.499337	12.704505	11.151741	13.726987	14.150235	12.042804	10.153908
50%	14.547655	13.802588	12.324697	14.768760	15.066293	13.137733	11.481316
75%	15.606746	15.057733	13.601164	15.953466	16.179981	14.411716	12.877707
max	inf	inf	inf	inf	inf	inf	inf

8 rows × 1240 columns

In [58]:
```
for eps in [6, 7, 8, 9, 10, 11, 12, 14]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Number of clusters: {}".format(len(np.unique(labels))))
    print("Cluster sizes: {}".format(np.bincount(labels + 1)))
```

```
eps=6
Number of clusters: 2
Cluster sizes: [1236    4]

eps=7
Number of clusters: 2
Cluster sizes: [1181   59]

eps=8
Number of clusters: 3
Cluster sizes: [1036  201    3]

eps=9
Number of clusters: 3
Cluster sizes: [816 420   4]

eps=10
Number of clusters: 2
Cluster sizes: [577 663]

eps=11
Number of clusters: 2
Cluster sizes: [349 891]

eps=12
Number of clusters: 2
Cluster sizes: [ 166 1074]

eps=14
Number of clusters: 2
Cluster sizes: [  21 1219]
```

- For lower `eps` all images are labeled as noise.
- For `eps=7` we get many noise points and many small clusters.
- For `eps=8` and `eps=9` we get many noise points but we also get one large cluster and a few smaller clusters.
- Starting `eps=10` we get one big cluster and noise points.
- There is never more than one large cluster suggesting that all the images are more or less equally similar/dissimilar to the rest.

Noise images identified by DBSCAN

In [59]:
```
dbscan = DBSCAN(eps=14, min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

```
Unique labels: [-1  0]
```

In [60]:
```
print_dbscan_noise_images(X_people, y_people, dbscan, labels)
```

In [61]:
```
plt.show()
```

- We can guess why these images are noise images. There are odd angles, cropping, sun glasses, hands near faces etc.

Let's examine DBSCAN clusters.

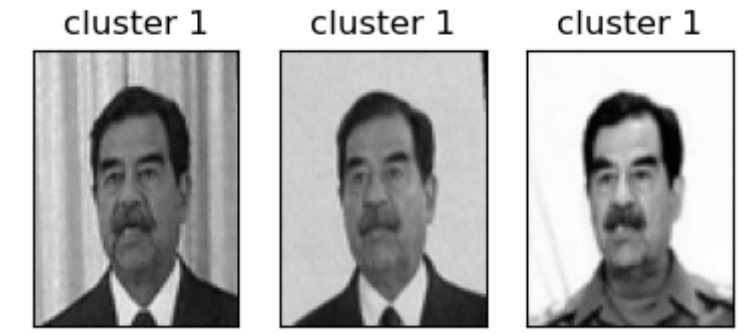
In [62]:
```
dbscan = DBSCAN(min_samples=3, eps=8)
labels = dbscan.fit_predict(X_pca)
print("Number of clusters: {}".format(len(np.unique(labels))))
print("Cluster sizes: {}".format(np.bincount(labels + 1)))
```

```
Number of clusters: 3
Cluster sizes: [1036  201    3]
```

- Some clusters correspond to people with distinct faces and facial expressions.
- It's also capturing orientation of the face.

In [63]:
```
print_dbscan_clusters(X_people, y_people, labels)
```

In [64]:
```
1 plt.show()
```



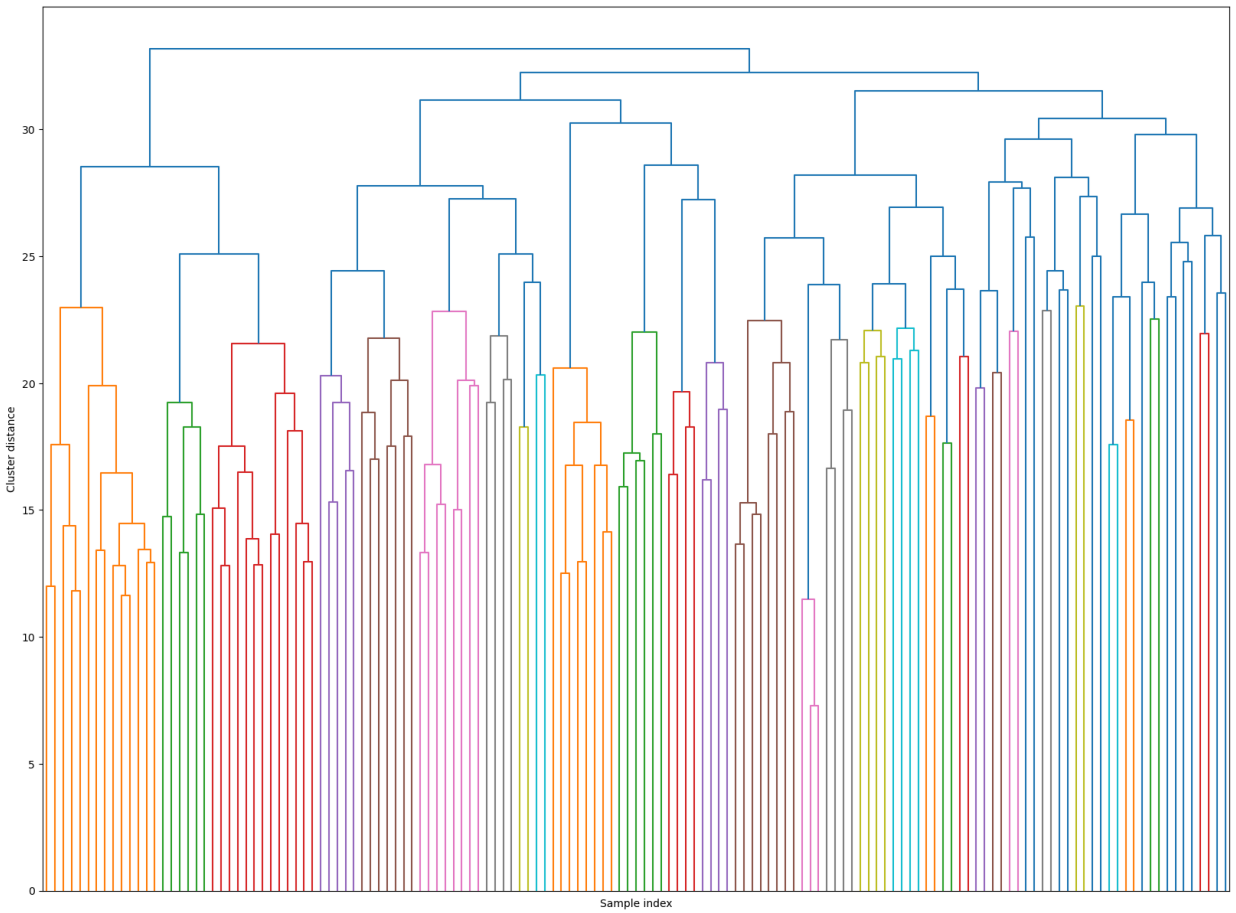
(Optional) Clustering faces with hierarchical clustering

Let's examine the dendrogram.

In [65]:
```
1 Z = ward(X_pca)
2 plt.figure(figsize=(20, 15))
3 dendrogram(Z, p=7, truncate_mode="level", no_labels=True)
4 plt.xlabel("Sample index")
5 plt.ylabel("Cluster distance");
```

In [66]:
```
1 cluster_labels = fcluster(Z, 40, criterion="maxclust")  # let's get fla
```

In [67]:
```
hand_picked_clusters = [2, 3, 6, 29, 30, 36, 38]
print_hierarchical_clusters(
    X_people, y_people, people.target_names, cluster_labels, hand_picke
)
plt.show()
```



Final comments, summary, and reflection

Take-home message

- We saw three methods for clustering: K-Means, DBSCAN, and hierarchical clustering.
- There are many more clustering algorithms out there which we didn't talk about. For example see [this overview of clustering methods (https://scikit-learn.org/stable/modules/clustering.html#overview-of-clustering-methods)](https://scikit-learn.org/stable/modules/clustering.html#overview-of-clustering-methods).
- Two important aspects of clustering
  - Choice of distance metric
  - Data representation
- Choosing the appropriate number of clusters for a given problem is quite hard.
- A lot of manual interpretation is involved in clustering.

A few comments on clustering evaluation

- If you know the ground truth, you can use metrics such as [adjusted random score (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html)](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html) or [normalized mutual information score (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.normalized_mutual_info_score.html)](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.normalized_mutual_info_score.html).
- We can't use accuracy scores.
  - Because the labels themselves are meaningless in clustering.
- Usually ground truth is not available, and if it is available we would probably go with supervised models.

- The silhouette score works for different clustering methods and it can give us some intuition

A couple of ways to evaluate clustering:

- Using *robustness-based* clustering metrics
- The idea is to run a clustering algorithm or a number of clustering algorithms after adding some noise to the data or using different parameter settings and comparing outcomes.
- If many models, perturbations, and parameters are giving the same result, the clustering is likely to be trustworthy.
- That said, even though all clustering models give similar results, the clusters might not capture the aspect you are interested in.
- So you cannot really avoid manual analysis!!

Resources

- Check out this nice comparison of sklearn clustering algorithms (https://scikit-learn.org/stable/modules/clustering.html#overview-of-clustering-methods).
- DBSCAN Visualization (https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/)
- Clustering with Scikit with GIFs (https://dashee87.github.io/data%20science/general/Clustering-with-Scikit-with-GIFs/)